Precise and Scalable Constraint-Based Type Inference for Incomplete Java Code Snippets in the Age of Large Language Models

by

Yiwen Dong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Song Wang Associate Professor, Lassonde School of Engineering, York University
Supervisor(s):	Chengnian Sun Associate Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Member:	Ondřej Lhoták Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Member:	Pengyu Nie Assistant Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Werner Dietl Associate Professor, Electrical and Computer Engineering, University of Waterloo

Author's Declaration

Statement of Contributions

Earlier versions of the work in this thesis have been published as follows:

- **SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets** (Chapter 3). Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. In Proceedings of the International Conference on Software Engineering (ICSE 2022), p. 1982–1993. Available: https://doi.org/10.1145/3510003.3510061

- **Beyond Memorization: Evaluating the True Type Inference Capabilities of LLMs for Java Code Snippets** (Chapter 4). Yiwen Dong, Zhenyang Xu, Yongqiang Tian, and Chengnian Sun. Under submission.

- **Scitix: Scalable Constraint-Based Type Inference for Code Snippets with Unknown Types** (Chapter 5). Yiwen Dong, Zhenyang Xu, Yongqiang Tian, Edward Lee, Ondřej Lhoták, and Chengnian Sun. Under submission.

I was the primary author of all the research presented in this thesis. My contributions included: (1) developing the research ideas, (2) designing and conducting experiments, (3) analyzing and interpreting results, and (4) drafting the manuscripts. My co-authors assisted with refining research ideas, suggesting additional experiments, and providing valuable feedback on the manuscripts.

Abstract

Online code snippets are prevalent and are useful for developers. These snippets are commonly shared on websites such as Stack Overflow to illustrate programming concepts. However, these code snippets are frequently incomplete. In Java code snippets, type references are typically expressed using simple names, which can be ambiguous. Identifying the exact types used requires fully qualified names typically provided in import statements. Despite their importance, such import statements are only available in 6.88% of Java code snippets on Stack Overflow. To address this challenge, this thesis explores constraint-based type inference to recover missing type information. It also proposes a dataset for evaluating the performance of type inference techniques on Java code snippets, particularly large language models (LLMs). In addition, the scalability of the initial inference technique is improved to enhance applicability in real-world scenarios.

The first study introduces SnR, a constraint-based type inference technique to automatically infer the exact type used in code snippets and the libraries containing the inferred types, to compile and therefore reuse the code snippets. Initially, SnR builds a knowledge base of APIs, *i.e.*, various facts about the available APIs, from a corpus of Java libraries. Given a code snippet with missing import statements, SnR automatically extracts typing constraints from the snippet, solves the constraints against the knowledge base, and returns a set of APIs that satisfies the constraints to be imported into the snippet. When evaluated on the StatType-SO benchmark suite, which includes 267 Stack Overflow code snippets, SnR significantly outperforms the state-of-the-art tool Coster. SnR correctly infers 91.0% of the import statements, which makes 73.8% of the snippets compilable, compared to Coster's 36.0% and 9.0%, respectively.

The second study evaluates type inference techniques, particularly of LLMs. Although LLMs demonstrate strong performance on the StatType-SO benchmark, the dataset has been publicly available on GitHub since 2017. If LLMs were trained on StatType-SO, then their performance may not reflect how the model would perform on novel, real-world code, but rather result from recalling examples seen during training. To address this, this thesis introduces ThaliaType, a new, previously unreleased dataset containing 300 Java code snippets. Results reveal that LLMs exhibit a significant drop in performance when generalizing to unseen code snippets, with up to 59% decrease in precision and up to 72% decrease in recall. To further investigate the limitations of LLMs in understanding the execution semantics of the code, semantic-preserving code transformations were developed. Analysis showed that LLMs performed significantly worse on code snippets that are syntactically different but semantically equivalent. Experiments suggest that the strong performance of

LLMs in prior evaluations was likely influenced by data leakage in the benchmarks, rather than a genuine understanding of the semantics of code snippets.

The third study enhances the scalability of constraint-based type inference by introducing Scitix. Constraint-solving becomes computationally expensive using a large knowledge base in the presence of unknown types (*e.g.* user-defined types) in code snippets. To improve scalability, Scitix represents certain unknown types as `Any`, ignoring such types during constraint solving. Then an iterative constraint-solving approach saves on computation and skips constraints involving unknown types. Extensive evaluations show that the insights improve both performance and scalability compared to SnR. Specifically, Scitix achieves F1-scores of 96.6% and 88.7% on StatType-SO and ThaliaType, respectively, using a large knowledge base of over 3,000 jars. In contrast, SnR consistently times out, yielding F1-scores close to 0%. Even with the smallest knowledge base, where SnR does not time out, Scitix reduces the number of errors by 79% and 37% compared to SnR. Furthermore, even with the largest knowledge base, Scitix reduces error rates by 20% and 78% compared to state-of-the-art LLMs.

This thesis demonstrates the use of constraint-based type inference for Java code snippets. The proposed approach is evaluated through a comprehensive analysis that contextualizes its performance in the current landscape dominated by LLMs. The ensuing system, Scitix, is both precise and scalable, enhancing the reusability of Java code snippets.

Acknowledgements

I would like to thank the many individuals who supported me through my Ph.D. journey. This would not have been possible without them.

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Chengnian Sun. During my years of study, you taught me critical thinking, writing, problem-solving, and much more. Your valuable feedback and guidance helped me succeed in this journey. And I appreciate the academic freedom I have been given to explore interesting topics. I am truly lucky to have met you.

I am also grateful to my committee members, Prof. Song Wang, Prof. Ondřej Lhoták, Prof. Pengyu Nie, and Prof. Werner Dietl for your time and your feedback. In addition, I like to thank Prof. Yizhou Zhang for serving on my comprehensive committee. I am grateful to my co-authors for their countless valuable suggestions, and I would like to thank Prof. Yongqiang Tian for his helpful comments.

I would like to thank all the professors in SWAG for all the insightful conversations and amusing stories over the years, Prof. Mei Nagappan, Prof. Mike Godfrey, Prof. Shane McIntosh, and Prof. Pengyu Nie. I would also like to thank the numerous SWAG labmates and lab adjacent friends. Apologies for any names I missed; names are hard for me. Thank you to Wenhan (Cosmos), Zheyang (Charles), Jack, Edward, Yongqiang (Victor), Shameer, Vikram, Yaxin, Xueyan, Zhenyang, Gaosen, Boren, Partha, Daniel, Farshad, Xueyao (Eve), Mengxiao (Max), Aryan, Raymond, Yelizaveta (Liza), Genyi (Gen), Gaoshuai (Albert), Mahtab (Mattie), Nimmi, Mahmoud, Amelia, Yiran, Noble, Wanying, Xintong, Xiang (Echo), Zhifeng (Anthony), Yimu, Amber, Bihui, Ruizhe, and many more.

I am deeply grateful to all my friends who helped me through difficult times. Thank you to Kyle for the moral support, Emily for the adventures and hugs, Rish, Erick, and Sarah for finding me, and Ben, Hidi, Jeremy, Chase, Julie, Max, and Erica for all the quarantine shenanigans. Thank you to Justin, Andrew, Joon, and Matt for always being there, and to my roommates Noah, Maulik, and Rodrigo for putting up with me. I am fortunate to have such wonderful friends.

Finally, I thank my family. My parents Xin Dong and Liehua Shi. They're a constant source of strength and motivation. Without their support, I might not have had the courage to set off on this journey to pursue a Ph.D. I am also grateful to my uncle and aunts, grandparents, and cousins as well for always supporting us.

I especially want to remember my father, Xin, who sadly could not see me complete this Ph.D. journey. He was the foundation of our family. His love, sacrifices, and the values he

instilled in me continue to guide and support me every day. He taught me the importance of hard work, determination, and never wasting time, lessons worth sharing.

Dedication

This thesis is dedicated to all who have left their ripples in the vast ocean of life, those who came before, those who are here now, and those yet to come.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code snippets are widely reused by developers, yet they often lack crucial dependency information, making their reuse difficult. These snippets, which are incomplete fragments of code frequently found on platforms such as Stack Overflow, typically include only the minimal function or variable declarations necessary to convey a programming concept. Consequently, they often exclude the surrounding context. For example, Java code snippets commonly omit class declarations, import statements, and other essential dependency information required for successful compilation and execution. Developers who wish to reuse these snippets must therefore manually identify the relevant libraries and resolve the referenced types in order to integrate and execute the code successfully.

Currently, there are no precise, automated, and explainable techniques for recovering missing dependency information and import statements from such incomplete code snippets. This challenge is particularly pronounced in Java, where ambiguous simple names are commonly used to reference types from external libraries. Resolving this ambiguity requires adding fully qualified names (FQNs) through import statements, which are typically absent in snippets due to their incomplete nature. As a result, developers are often forced to search across the extensive Java ecosystem, which contains a vast number of available libraries, to identify the correct dependencies for successful snippet reuse.

Recent research has explored the use of type inference to automatically recover missing types and generate the corresponding import statements [1, 2, 3, 4, 5, 6, 7, 8]. By inferring types and resolving their FQNs, these methods can transform incomplete snippets into compilable code. However, earlier machine learning-based techniques have struggled with low performance [2, 3], often requiring developers to manually correct erroneous suggestions, undermining the goal of automation.

Recent advances in large language models (LLMs) have significantly improved the ability to infer missing information from incomplete code snippets [5, 7, 8]. LLMs, trained on massive corpora that include programming code and related natural language, can leverage their broad knowledge to predict likely types and generate appropriate import statements. These capabilities have raised hopes that LLMs may overcome the limitations of earlier machine learning approaches and provide more reliable automation for type inference on Java code snippets.

However, evaluations of LLM-based type inference for Java code snippets have relied on code snippets from Stack Overflow, using a benchmark dataset that has been publicly available on GitHub since 2017 [5, 7, 8]. This raises concerns about data leakage, as these models may have been trained on data overlapping with the evaluation set, potentially inflating their performance. Due to the proprietary nature of LLM training data, it is often impossible to conclusively assess the extent of such data leakage, making fair and accurate evaluation challenging.

Beyond evaluation concerns, Java's thriving ecosystem requires type inference techniques to have knowledge of a large number of libraries and types that may be used in code snippets. Unlike traditional software projects that typically depend on a limited set of known libraries, code snippets may draw from any part of a vast and continually expanding collection of Java libraries. The number of potential types increases with the size of the ecosystem. Consequently, type inference techniques must be both precise and scalable to handle the large and diverse set of libraries and types involved in snippet reuse.

1.1 Overview

This thesis focuses on type inference for incomplete Java code snippets.

Thesis Statement

This thesis demonstrates that constraint-based type inference provides a precise and scalable solution for recovering missing import statements from incomplete code snippets, outperforming prior machine learning-based approaches and state-of-the-art LLMs.

This thesis consists of three studies that build upon one another. The first introduces constraint-based type inference using SnR, which outperforms prior machine learning-based techniques. The second evaluates state-of-the-art LLM-based techniques using SnR as one of the baselines to assess true LLM performance. The third proposes Scitix, which extends constraint-based inference to preserve precision while improving scalability.

Constraint-Based Type Inference for Incomplete Java Code Snippets. The first study introduces SnR to precisely infer the types used in incomplete Java code snippets. SnR tackles the challenges with a lack of dependency information and a lack of import statements by leveraging constraints. Given a code snippet with missing import statements, SnR automatically extracts typing constraints from the snippets. These constraints are then solved using a constraint solver with a pre-built knowledge base of types, collected from various Java libraries. The solver identifies the set of types that satisfy the constraints which can be used to suggest the necessary import statements for the code snippet.

SnR was evaluated on a benchmark dataset of 267 Stack Overflow code snippets, named StatType-SO, and achieved substantial improvements over the state-of-the-art tool Coster. Specifically, SnR correctly inferred 91.0% of the import statements, compared to Coster's 36.0%.

This study was published at the 44th International Conference on Software Engineering (ICSE'22) [4].

Evaluating the True Type Inference Performance of LLMs. The second study investigates the potential impact of data leakage on LLMs when applied to type inference. Since SnR's publication, advances in LLMs allow developers to directly ask LLMs to infer import statements on code snippets with promising results [5, 7, 8]. However, given that state-of-the-art LLMs are trained on massive datasets from the internet, including platforms such as Stack Overflow and GitHub, concerns arise about data leakage. This is particularly relevant because the StatType-SO dataset has been publicly available on GitHub since 2017.

To assess if LLMs' true type inference performance was misrepresented due to data leakage, a two-fold investigation is conducted. First, Thalia, a testing framework for type systems, was repurposed to randomly generate 300 novel code snippets, ensuring they were previously unseen by the LLMs. These code snippets are referred to as ThaliaType. When evaluated on ThaliaType, LLMs exhibited a performance decrease of up to 67.2% compared to their performance on StatType-SO. The performance decrease on LLMs with unknown training data is consistent with the baseline LLM with known data leakage and far more than the SnR baseline. Second, semantic preserving transformations were applied to both StatType-SO and ThaliaType code snippets to test LLMs' generalization capabilities despite syntactic differences. Isolated transformations showed that LLMs are able to generalize to code snippets with syntactic differences. However, when transformations are combined, all tested LLMs showed consistent, significant performance decreases in only StatType-SO code snippets.

These findings suggest that future evaluations of LLM-based type inference techniques should consider data leakage and evaluate using generated and transformed code snippets to ensure the tool is able to generalize to unseen code snippets.

This study was submitted to ACM Transactions on Software Engineering and Methodology (TOSEM) and is undergoing major revision.

Scalable Constraint-Based Type Inference with Unknown Types. The third study presents Scitix to improve the scalability of constraint-based type inference when unknown types are present. While `SnR` is efficient when the knowledge base contains all the types that are used in the code snippet, in practice, there are often unknown types (*e.g.*, user-defined types) present in code snippets, which makes `SnR` inefficient. To address this challenge, Scitix adds a special `Any` type to represent the known unknown types present in the code snippet and employs an iterative approach to add the constraints unrelated to unknown types, thereby maintaining scalability.

Scitix was evaluated using seven knowledge bases comprising up to 3,049 Java libraries. It consistently outperformed `SnR` across all knowledge base sizes. Specifically, Scitix achieved F1-scores of 96.6% on Stack Overflow snippets and 88.7% on ThaliaType with the largest knowledge base. In contrast, `SnR` consistently times out, resulting in near-zero F1 scores. Even with the smallest knowledge base, Scitix reduces the number of errors by 79% and 37% compared to `SnR`. Additionally, Scitix reduces error rates by 20% and 78% compared to state-of-the-art LLMs, even when using the largest knowledge base.

These results demonstrate Scitix's potential as a practical, scalable solution for constraint-based type inference in real-world code snippets.

This study was submitted to ACM Transactions on Software Engineering and Methodology (TOSEM) and is awaiting decision.

1.2 Contribution

This thesis makes three main contributions to support code snippet reuse by automatically inferring missing import statements.

Constraint-Based Type Inference. We propose `SnR`, a novel constraint-based approach that automatically and precisely infers types and creates import statements for incomplete Java code snippets. Our comprehensive evaluation using StatType-SO demonstrates that SnR greatly outperforms the state-of-the-art tool in type inference.

Evaluation Against LLMs. Through a novel application of Thalia, we introduce ThaliaType, a novel benchmark of unseen code snippets designed to mitigate potential data leakage when evaluating LLM-based type inference. Evaluation using ThaliaType and transformed code snippets reveal that LLMs may be affected by data leakage, exhibiting reduced generalizability on StatType-SO.

Scalable Constraint-Based Type Inference With Unknown Types. We further proposed Scitix, a scalable approach for type inference in the presence of unknown types through the introduction of the `Any` type and an iterative approach to add constraints. Scitix outperforms both SnR and LLMs in StatType-SO and ThaliaType code snippets.

1.3 Organization

The remainder of the thesis details the three studies described above. §2 introduces the necessary background, including the structure of Java programs, an overview of Datalog as used for constraint solving, and the use of LLMs in software engineering. §3 presents SnR, a constraint-based type inference approach. §4 evaluates LLMs to assess their true type inference performance. §5 improves SnR by introducing Scitix, a scalable solution for type inference in the presence of unknown types. Finally, §6 concludes the thesis and outlines directions for future work.

Chapter 2

Background

Type inference on incomplete Java code snippets primarily involves inferring the types of the simple names used in the snippet. While this thesis focuses on Java, the concepts presented are applicable to a broad spectrum of programming languages. In this chapter, §2.1 briefly introduces the essential Java concepts necessary to understand our approach. Subsequently, §2.2 introduces the notation and system that are used to represent and solve the constraints. §2.3 then gives some background information on LLMs and their use in software engineering.

2.1 Java Program

In a typical Java program [9], there is a set of compilation units, each of which is a Java source file. Each Java source file contains a set of import statements, defines one top-level class, and may include any number of inner classes. The import statements are used to specify the FQNs for the simple names used in class or classes within the source file. Generally, a Java class has the following components which are used by `SnR` for type inference.

Name Each type (*e.g.*, class, interface, annotations) has a *fully qualified name*, which is the combination of its *package name* (may be empty) and *simple name*. Syntactically, an FQN is a sequence of *simple names* joined by dots.
Super Class Each class has a single *super class* except the class `java.lang.Object` (`Object` for short). The default super class of a class is `Object` if the `extends` declaration is absent.

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Name} \mid \mathit{Literal} \mid \texttt{this} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
 & & \mid\ \texttt{(}\ \mathit{Type}\ \texttt{)}\ \mathit{Expr} \mid \mathit{Expr}\ \texttt{[}\mathit{Expr}\texttt{]} \\
 & & \mid\ \mathit{Expr}\ .\ \mathit{SimpleName} \\
 & & \mid\ \mathit{Expr}\ \texttt{instanceof}\ \mathit{ReferenceType} \\
 & & \mid\ \mathit{Expr}\ .\ \mathit{SimpleName}\ \texttt{(}\{\mathit{Expr}\}\texttt{)} \\
 & & \mid\ \texttt{new}\ \mathit{ClassType}\ \texttt{(}\{\mathit{Expr}\}\texttt{)} \\
 & & \mid\ \texttt{new}\ \mathit{Type}\ \texttt{[}\ \mathit{Expr}\ \texttt{]} \\
\mathit{Name} & ::= & \mathit{FQN} \mid \mathit{SimpleName} \\
\mathit{FQN} & ::= & \mathit{Name}\ .\ \mathit{SimpleName} \\
\mathit{Literal} & ::= & \texttt{null} \mid \mathit{NumberLiteral} \mid \mathit{StringLiteral} \\
\mathit{Op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{>} \mid \texttt{==} \mid \texttt{>=} \mid \texttt{!=} \\
\mathit{Type} & ::= & \mathit{PrimitiveType} \mid \mathit{ReferenceType} \\
\mathit{ReferenceType} & ::= & \mathit{ClassType} \mid \mathit{ArrayType} \\
 & & \mid\ \mathit{ParameterizedType} \\
\mathit{PrimitiveType} & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{boolean} \\
\mathit{ClassType} & ::= & \mathit{Name} \\
\mathit{ArrayType} & ::= & \mathit{Type}\ \texttt{[ ]} \\
\mathit{ParameterizedType} & ::= & \mathit{ClassType}\ \texttt{<}\{\mathit{ClassType}\}\texttt{>}
\end{array}
$$

Figure 2.1: Part of a simplified grammar of the expressions in Java. {*} denotes that the enclosed term occurs zero or more times.

Interfaces Each class can have a set of *interfaces* as supertypes.
Annotations Each class can have a set of *annotations*.
Fields Each class can have a set of *fields*. Each field has a type and a name, and can be optionally initialized with an expression.
Methods Each class has a set of *methods*. Each method has a *name*, an optional sequence of parameters, and a return type. Methods contain statements and expressions in them.

Figure 2.1 shows a simplified version of the grammar of expressions in Java. We will use this grammar and the language constructs listed above to illustrate the type inference techniques in this thesis.

2.2 Datalog

Our techniques (SnR and Scitix) leverage Datalog, a declarative logic programming language that emerged from database systems in the 1980s [10]. In recent years, Datalog has found use in a whole range of applications [11, 12] in particular program analysis [13, 14, 15, 16, 17]. Numerous implementations of Datalog have emerged over the years [18, 19, 20, 21, 22]. This thesis utilizes the Soufflé Datalog engine [20], chosen for its high performance on static analysis workloads [23].

A Datalog grammar is shown in Figure 2.2a. A Datalog program consists of a list of facts and rules, representing sets of relations. Facts are known relations given to a Datalog program and rules are used to derive relations using facts and other rules as specified by the program. Relations can be queried to return all satisfying constants.

Figures 2.2b–2.2e use a dependency graph as an example to illustrate the various facets of a Datalog program. The dependency graph in Figure 2.2b has three nodes `a`, `b`, and `c` where `a` depends on `b` and `b` depends on `c`. The information of this dependency graph can be represented as the Datalog facts shown in Figure 2.2d. To recursively query the reachable node pairs in this dependency graph, we write a demonstration Datalog program consisting of two rules as shown in Figure 2.2c. When we query for the `reachable` relation, all the reachable node pairs are returned by a Datalog solver as seen in Figure 2.2e, *i.e.*, `(a,b)`, `(a,c)`, and `(b,c)`.

As will be shown in §3 and §5, constraint-based type inference involves extracting constraints from code snippets and refining facts from a pre-built knowledge base. These constraints and facts are then provided to a Datalog system to get sets of FQNs that satisfy all the given constraints.

2.3 Large Language Models

Large language models are machine-learning models trained on vast amounts of text to perform a wide range of generative tasks [24, 25, 26, 27]. The most capable state-of-the-art LLMs are based on the transformer architecture [28], which enables efficient training on massive datasets, typically using text or code scraped from the internet [24, 25, 26].

In software engineering, LLMs have been applied to tasks such as automated code repair [29, 30, 31, 32, 33, 34, 35, 36], code generation [37, 38, 39, 40], code refactoring [41, 42], and code completion [43, 44, 45]. Naturally, prior type inference techniques for Java

$$\begin{array}{rcl}
\textit{Program} & ::= & \textit{Fact Program} \mid \textit{Rule Program} \mid \varepsilon \\
\textit{Fact} & ::= & \textit{relation}\ \texttt{(}\ \{\textit{constant}\}\ \texttt{)}\ \texttt{.} \\
\textit{Rule} & ::= & \textit{Atom}\ \texttt{:-}\ \textit{NAtom}\ \{\textit{NAtom}\}\ \texttt{.} \\
\textit{NAtom} & ::= & \textit{Atom} \mid \texttt{!}\ \textit{Atom} \\
\textit{Atom} & ::= & \textit{relation}\ \texttt{(}\ \{\textit{Term}\}\ \texttt{)} \\
\textit{Term} & ::= & \textit{constant} \mid \textit{variable}
\end{array}$$

(a) Simplified Datalog grammar. The terms *relation* and *variable* are names for defining relationships declaring and referencing variables. *constant* is either a numerical or string literal. {*} denotes that the enclosed term occurs zero or many times.

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \end{tikzcd}$$

(b) Dependency graph.

```
reachable(s,t) :- node(s) node(t)
                  edge(s, t).
reachable(s,t) :- node(s) node(m) node(t)
                  edge(s, m)
                  reachable(m,t).
```

(c) Recursive Datalog rules defining reachability.

```
node("a").
node("b").
node("c").
edge("a","b").
edge("b","c").
```

(d) Datalog facts representing the dependency graph.

```
reachable("a","b").
reachable("a","c").
reachable("b","c").
```

(e) Successful reachable query given the facts in Figure 2.2d.

Figure 2.2: Datalog grammar and a Datalog example program.

code snippets have not only used LLMs as baselines for comparison [5, 8], but have also incorporated LLMs in an iterative workflow for type inference [7].

However, a serious concern when evaluating the performance of LLMs is data leakage [46], which occurs when a model is evaluated on data it has seen during training, leading to inflated performance that may not generalize to unseen data. Since LLMs are often trained on data scraped from public sources, they may inadvertently ingest datasets used for evaluation. For example, benchmarks such as Defect4J [47] (for automated code repair), MBPP [48], and HumanEval [49] (for code generation) have come under increasing scrutiny for potential leakage [50, 51, 52].

Similarly, type inference techniques for Java code snippets are often evaluated using StatType-SO, a dataset collected from the Stack Overflow website. LLMs trained on Stack Overflow data may gain unintended context, such as replies containing expected import statements, which can artificially inflate performance. Compounding this issue, we observed that StatType-SO, including the manually repaired import statements, is publicly available on GitHub (§4.4.2), further motivating our investigation into LLM type inference performance §4.

Chapter 3

SnR: Constraint-Based Type Inference for Incomplete Code Snippets

This chapter introduces SnR, a technique that takes an incomplete code snippet and constructs a compilation unit through constraint-based type inference. This chapter demonstrates that by leveraging both syntactic and semantic relationships within the snippet, SnR enables precise inference of missing types, which can then be used to repair the code snippet.

3.1 Introduction

Although many prior approaches aim to assist developers in repairing incomplete code snippets, they all fall short of addressing all aspects of the problem required for effective reuse. We identify three core technical challenges that must be addressed.

Challenge 1: Lack of Import Statements. In Java, a class type has a simple name (*e.g.*, `Date`) and a package name (*e.g.*, `java.util`); its FQN is the combination of its package name and simple name (*e.g.*, `java.util.Date`), which uniquely identifies the class type. To make Java code concise, the simple name of a class type can be used in Java code directly, and the class type's FQN is declared by an import statement. With the help of import statements a Java compiler can identify the exact class type from a simple name during compilation. Previous work [53] showed that only 6.88% of Java code snippets on Stack Overflow included import statements that specified the FQNs of the types used in the code snippets. One example is Figure 3.1; the Java compiler is not able to infer the

```java
DateFormat formatter = new SimpleDateFormat("mm/dd/yyyy");
Date someDate = new Date();
Date today = Calendar.getInstance().getTime();
try {
    someDate = formatter.parse("06/22/2010");
} catch(ParseException pe) {
    System.out.println("Parser Exception");
}
int days = Days.daysBetween(new DateTime(someDate), new DateTime(today)).getDays();
System.out.println(" Days Between " + someDate + " : " + today + " - " + days);
```

Figure 3.1: Formatted code snippet in Stack Overflow post #3329469.

FQNs from simple names such as `Date`, `Days`, and `DateFormat`, because the code snippet does not have any import statements.

Challenge 2: Lack of Library Dependencies. A code snippet usually does not carry information about the libraries. These libraries are needed for the Java compiler to compile the code snippet as they contain the type definitions used in the snippet. The example in Figure 3.1 required the Joda-Time library. Note that even correct FQNs do not guarantee that we can find the correct library to depend on, because different libraries of different purposes may contain types with the same FQN. For example, the Android runtime library and the Java runtime library both define a class for `java.text.DateFormat`.

Challenge 3: Combinatory Candidates. To compile a code snippet, we need to have the correct FQN for each simple name, and the correct library for each FQN. However, a simple name may correspond to multiple different FQNs, and each FQN may correspond to multiple different libraries. The search space is all the combinations of candidates for each simple name, which defines a computationally expensive problem. For example, in Figure 3.1, the simple name `Date` has five matching classes in the Java Development Kit (JDK) alone; in total, the search space for this code snippet is 384 different combinations of classes from the JDK and five other popular Java libraries used in our benchmarks.

Prior Work. Existing techniques attempted to address type inference for code snippets in different manners: CSNIPPEX is based on a set of heuristics [53]; Baker extracts constraints from code snippets and uses a naive constraint solving algorithm to infer FQNs [1]; both StatType [2] and Coster [3] build statistical models from existing compilable source code to predict FQN for code snippets. However, these techniques do not address all three challenges and suffer from inherent imprecision of the used heuristics [53], proposed constraint solving algorithm [1], or the trained statistical models [2, 3]; *especially neither of them tackles Challenge 2 when multiple libraries contain different types with the same* *simple names.*

Our Approach. We propose SnR, a novel, constraint-based approach to automatically, and precisely infer FQNs and required libraries to compile and reuse code snippets. SnR builds a knowledge base from available libraries by extracting facts of the types defined in these libraries, *e.g.*, fields, methods, signatures, and inheritance relations. Given an incomplete code snippet, SnR extracts constraints from the code snippet that capture the relation between types used in the snippet; then SnR resolves these constraints by querying the knowledge base, and outputs *a ranked list of solutions* where each solution is a set of types that satisfy the constraints and likely make the code snippet compile.

Compared to the prior work based on either heuristics or statistics, SnR leverages the type system built into programming languages and models the problem of inferring types for incomplete code snippets as a constraint satisfaction problem (CSP). Without being affected by the randomness, approximation, and un-interpretability from which prior work suffers, SnR can deterministically, and precisely infer types with explicit explanations of how and why the types are inferred.

We thoroughly evaluated SnR against the state-of-the-art tool, Coster [3]. We used an established benchmark called *StatType-SO* consisting of 267 code snippets manually collected from Stack Overflow posts [2]. SnR significantly outperformed Coster in terms of accuracy of type inference and library recommendation: ① In the task of inferring types for API elements (including simple names, field accesses, and method calls as defined in [3]), SnR achieves a high precision of 98.20% and recall of 79.66% compared to a precision of 66.35% and recall of 66.35% by Coster. ② In the task of inferring the import statements required for compiling code snippets, SnR is able to correctly infer 91.0% of the import statements compared to 36.0% by Coster. ③ SnR can accurately recommend libraries for snippets with an F_1 score of 0.82 compared to 0.53 by Coster. Notably, SnR recommended the exact libraries for 183 of the 267 snippets, compared to Coster with 34. ④ As a result of the high accuracy in type inference and library recommendation, SnR can make 73.8% of the code snippets compilable in total compared to 9.0% by Coster.

Contribution. We make the following major contributions.

- **Novelty** We proposed SnR, a novel constraint-based approach to automatically, and precisely infer FQNs, recommend libraries, and create import statements for code snippets.
- **Soundness** and **Significance** We conducted extensive evaluations on real-world code snippets in StatType-SO and the results demonstrate that SnR significantly outperforms the state of the art in various type inference tasks.

Table 3.1: The top-3 candidates output by SnR for Figure 3.1. The first row is the correct solution. The *FQNs in italics* are implemented in multiple libraries.

#	Name	Library	Fully Qualified Name
1	DateFormat	*jdk*	*java.text.DateFormat*
	SimpleDateFormat	*jdk*	*java.text.SimpleDateFormat*
	Date	*jdk*	*java.util.Date*
	ParseException	jdk	java.text.ParseException
	DateTime	joda	org.joda.time.DateTime
	Days	joda	org.joda.time.Days
2	DateFormat	*android*	*java.text.DateFormat*
	SimpleDateFormat	*android*	*java.text.SimpleDateFormat*
	Date	*android*	*java.util.Date*
	ParseException	android	android.net.ParseException
	DateTime	joda	org.joda.time.DateTime
	Days	joda	org.joda.time.Days
3	DateFormat	*jdk*	*java.text.DateFormat*
	SimpleDateFormat	*jdk*	*java.text.SimpleDateFormat*
	Date	*jdk*	*java.util.Date*
	ParseException	gwt	org.w3c.flute.parser.ParseException
	DateTime	joda	org.joda.time.DateTime
	Days	joda	org.joda.time.Days

- **Verifiability** We made a replication package available at `https://doi.org/10.5281/zenodo.5843327`

3.2 Motivating Example

We use the code snippet in Figure 3.1 as a motivating example to illustrate the main shortcomings of existing techniques, particularly in addressing Challenges 2 and 3.

Eclipse, a prevalent integrated development environment, has a powerful utility *Quick Fix* to fix common syntactical errors, repair partial statements, and insert missing import statements. However, Quick Fix is inadequate for inferring the types in Figure 3.1:

① It is imprecise due to its heuristic-based nature. For example, both `java.sql.Date` and `java.util.Date` are recommended by Quick Fix even though `Calender.getInstance()`

`.getTime()` only returns the latter.

② It only works with libraries and types on the class path, and cannot fix errors related to unknown types. For example, if the Joda-Time library is not on the class path, then Quick Fix cannot create import statements for `DateTime` which is a type defined in Joda-Time.

③ It cannot recommend new libraries to be added to the class path. Therefore, Quick Fix cannot recommend the Joda-Time library to developers.

Recent research attempted to address ① and ② using simple constraints [1] or more recently, statistics [2, 3]. We aimed to improve upon previous techniques and tackle ③. Previous solutions did not consider the same FQN being implemented by multiple libraries and did not recommend libraries as part of their inference. In the StatType-SO benchmark of six libraries alone, both the JDK and Android libraries provide implementations for many standard APIs. In the real world, we may want to include different versions of the same library *e.g.* for supporting both Java 8 and 12 APIs. In Table 3.1, we show a sample solution by SnR for the motivating problem Figure 3.1. The large number of FQNs with multiple implementations (shown in italics) demonstrates the need for a new technique that can recommend correct libraries. Including multiple libraries can lead to dependency conflicts and create serious runtime bugs [54]. To address the shortcomings of prior solutions, we strive to devise a technique to infer the correct FQNs from code snippets while minimizing the conflict of recommended libraries. *This is in contrast to Coster which recommends all libraries containing the inferred FQNs.*

3.3 Methodology

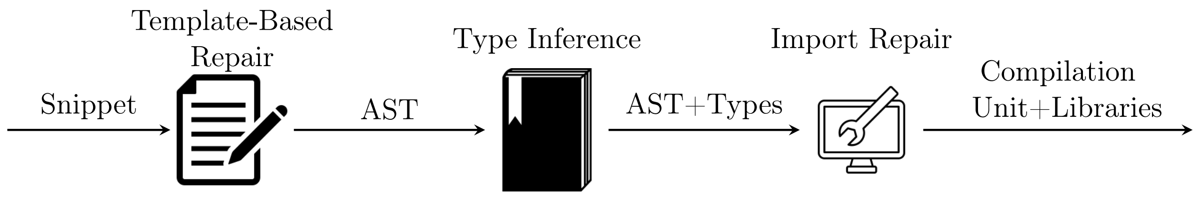


Figure 3.2: The overall workflow of SnR to repair a code snippet to a compilable compilation unit.

Figure 3.2 shows the overview of SnR. Given an incomplete code snippet as input, SnR aims to output a *compilable compilation unit* which is the input to a Java compiler (*i.e.*, a Java source file) and contains ① the code snippet, ② the necessary skeleton code

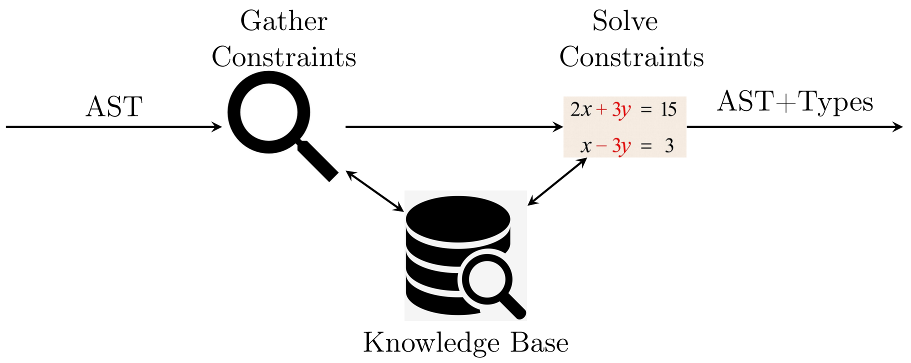


Figure 3.3: Type inference process in SnR.

(*e.g.*, a class definition and a method definition to enclose the code snippet) to make the compilation unit syntactically valid, and ③ inferred required import statements together with the libraries defining the imported types. To achieve this objective, the workflow of SnR has the following three major procedures.

Template-Based Repair. Given a code snippet c consisting of a list of statements, SnR first attempts to create a minimal code skeleton based on pre-defined templates to enclose c, so that the skeleton and c together form a syntactically valid compilation unit. Our approach is similar to that outlined by Terragni et al. [53]. After the repair, the Abstract Syntax Tree (AST) of the unit is generated for the following procedures.

Type Inference. Given the AST, SnR leverages the type inference engine to analyze and extract the constraints. These constraints encode the typing relations among types used in the AST. Then SnR refines the knowledge base, pre-built from a set of libraries, with concrete types to replace the generic types previously stored. Lastly, the refined knowledge base and constraints are given to Datalog to solve, giving us a list of solutions for the next step.

Import Repair. In the third step, SnR interprets the list of constraint-satisfying solutions, creates import statements, and inserts them into the code skeleton. To validate the results, SnR leverages the Java compiler to compile the resulting compilation unit. If the compilation succeeds, the compilation unit together with the import statements and the required libraries are output as the final result.

Type inference is the most critical step in SnR. Figure 3.3 describes the internal components of the type inference engine. In the remainder of this section, we describe these components in detail.

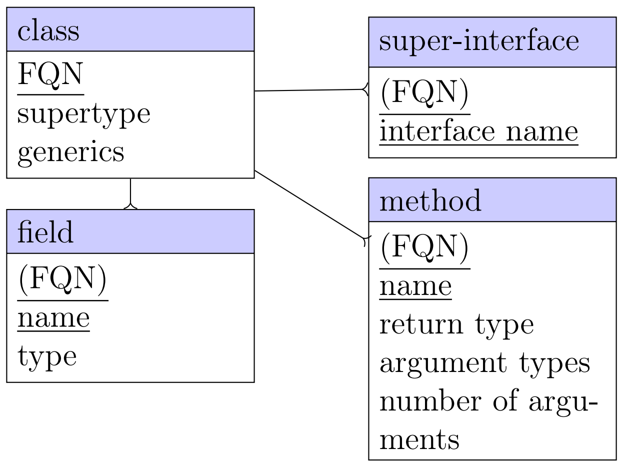


Figure 3.4: Simplified version of the knowledge base schema assuming there are no classes with the same FQN. Each box represents a table with a table name (in blue) and a number of column names. The underline denotes the primary key or keys that uniquely identify a row of data. The column names in parenthesis are foreign keys which are linked to primary keys in another table. An edge $\prec$ represents a one-to-many relationship between the two connected tables.

3.3.1 Knowledge Base

Given a set of libraries, we build the knowledge base which can be queried to resolve ambiguities (§3.3.3) when gathering constraints and solving constraints (§3.3.4).

Content

A simplified schema for our knowledge base is described in Figure 3.4. For each type, the knowledge base stores the FQN, supertype, super-interfaces, fields, and methods. Generic types are stored as is in the knowledge base. For example, `List.get()` in the knowledge base has the type `T` which will be refined before being given to Datalog when solving constraints, discussed in §3.3.4.

Note that the schema in Figure 3.4 is simplified to ease presentation with the assumption that there are no multiple classes with the same FQN. Moreover, library information (*e.g.*, which library defines a type) is also omitted in Figure 3.4. The real knowledge base in SnR handles both with additional database table columns.

Table 3.2: Query functions to retrieve types from knowledge base.

Query Function	Description
$\Omega_{simplename}$(name)	Returns the types with the given simple `name`
Ω_{fqn}(name)	Returns the types with the given fully qualified `name`
Ω_{field}(name)	Returns the types that have a field called the given `name`
Ω_{method}(name, num_args)	Returns the types defining a method with the given `name` which takes `num_args` number of parameters

Query Functions

We define the following query functions to retrieve information from the knowledge base. Each function represents a retrieval criterion and the parameters parameterize the criterion. Each function returns a set of types (*i.e.*, classes, interfaces, and annotations) that satisfy the specified retrieval criterion. Table 3.2 lists the query functions used by SnR.

Besides querying normal types, $\Omega_{simplename}$ has specialized support for querying inner types (*e.g.*, inner classes, inner interfaces). Note that an inner type can have multiple simple names. Consider an inner class `Builder` with an outer class `java.util.Calendar`. It is possible to reference this type with either of the two simple names `Calendar.Builder` or `Builder` using import statements `java.util.Calendar` or `java.util.Calendar.Builder` respectively.

$\Omega_{simplename}$ supports queries with different forms of simple names. In the example above, this query function can be called with either $\Omega_{simplename}$(`"Calendar.Builder"`) or $\Omega_{simplename}$(`"Builder"`).

3.3.2 Type Inference: Extracting Constraints

Given an AST, SnR traverses it and extracts the constraints capturing the relations among the types used in the AST. Specifically, SnR concentrates on *type elements* defined below,

Definition 3.3.1 (Type Elements)**.** *Type elements in an AST refer to the nodes that define new types or use types,* i.e.*, type declarations (*e.g.*, class, interface, and annotation declarations), explicitly used types, statements, and expressions.*

Note that prior work [3, 2] uses a different term *API elements*, which is a subset of type elements and focuses on only explicitly used types *i.e.* simple names, field accesses, and method calls. For high-precision type inference, SnR infers not only explicitly used types

Table 3.3: Constraints used by SnR. Each constraint specifies a property that a type variable τ should have, or a relation among multiple type variables.

Constraint	Description
`simplename(`τ`, name)`	τ has the given simple `name`
`fqn(`τ`, name)`	τ has the given FQN `name`
`field(`τ`, name, `τ_{field}`)`	τ has a field with given `name` of the type τ_{field}
`method(`τ`, name, `$\tau_{\vec{arg}}$`, `τ_{ret}`)`	τ has a method with given `name`, argument types $\tau_{\vec{arg}}$ and return type τ_{ret}
`paramtype(`τ`, `$\tau_{\vec{arg}}$`, `τ_{param}`)`	τ with parameter $\tau_{\vec{arg}}$ builds the parameterized type τ_{param}
`arraytype(`τ`, `τ_{arr}`)`	An array of τ is type τ_{arr}
`subtype(`τ_{parent}`, `τ_{child}`)`	τ_{child} can be implicitly converted to τ_{parent}
`extend(`$\tau, \tau_{\mathsf{super}}$`)`	τ extends τ_{super}
`interface(`τ`, `$\tau_{interface}$`)`	τ implements the interface of $\tau_{interface}$
`annotation(`τ`)`	τ is an annotation type
`innerclass(`τ`, `τ_{inner}`)`	τ is has an inner class type of τ_{inner}

but also implicitly used types in expressions, *e.g.* variables, method arguments, method returns. All the AST nodes used by SnR for type inference constitute type elements in this thesis.

Creating Type Variables

For each *type element*, SnR first creates one or more *type variables* to represent the types defined or used in the type element or in the components of the type element. For simplicity, we use τ to denote a type variable. Given the type element (a statement) below,

```
Date today = Calendar.getInstance().getTime();
```

SnR creates four type variables.

Type Variable	Description
τ_1	the type of `Date`
τ_2	the type of `Calendar`
τ_3	the return type of `Calendar.getInstance()`
τ_4	the return type of `Calendar.getInstance.getTime()`

Table 3.4: A subset of rules to extract constraints from type elements. Γ denotes an environment that maps an expression, e, to a type variable, τ; $\vec{x}$ denotes a vector of x where x can be an expression e, a type variable τ, or type t; $a \in \vec{x}$ denotes the check whether a is an element in the vector $\vec{x}$. $\Gamma(\vec{e})$ denotes a look up of every element in $\vec{e}$ on the environment Γ, and returns a vector of type variables with each variable corresponding to an expression in $\vec{e}$.[1]

Category	Name	Code	Type Variables	Constraint
Class	Declaration	$cls\ c\ ext\ t_1\ impl\ \vec{t_2}$	$c : \tau, t_1 : \tau_1, \vec{t_2} : \vec{\tau_2}$	$\texttt{extend}(\tau, \tau_1)$
				$\forall \tau_i \in \vec{\tau_2}, \texttt{interface}(\tau, \tau_i)$
Type	Type	t	$t : \tau$	$\texttt{simplename}(\tau, t)$
	Array Type	$t\ []$	$t : \tau_1, \Gamma(t[]) = \tau_2$	$\text{arraytype}(\tau_1, \tau_2)$
	Paramed Type	$t_1\langle\vec{t_2}\rangle$	$t_1 : \tau_1, \vec{t_2} : \vec{\tau_2}, \Gamma(t_1\langle\vec{t_2}\rangle) = \tau_3$	$\texttt{paramtype}(\tau_1, \vec{\tau_2}, \tau_3)$
Statement	If	if (e) $\{\vec{s}\}$	$\Gamma(e) : \tau$	$\texttt{subtype}(\tau, \text{"boolean"})$
	While	while (e) $\{\vec{s}\}$	$\Gamma(e) : \tau$	$\texttt{subtype}(\tau, \text{"boolean"})$
Expression	Assignment	$e_1\ =\ e_2$	$\Gamma(e_1) = \tau_1, \Gamma(e_2) = \tau_2$	$\texttt{subtype}(\tau_1, \tau_2)$
	Annotation	$@\ t$	$t : \tau$	$\text{annotation}(\tau)$
	Inner Class	$t\ .\ ct$	$t : \tau_1, \Gamma(t.ct) = ct : \tau_2$	$\texttt{simplename}(\tau_2, ct)$
				$\text{innerclass}(\tau_1, \tau_2)$
	Qualified Name	$n\ .\ sn$	$sn : \tau$	$\text{fqn}(\tau, n.sn)$
	Field	$e\ .\ f$	$\Gamma(e) = \tau_1, \Gamma(e.f) = \Gamma(f) = \tau_2$	$\text{field}(\tau_1, f, \tau_2)$
	Method	$e_1\ .\ m\ (\vec{e_2})$	$\Gamma(e_1) = \tau_1, \Gamma(\vec{e_2}) = \vec{\tau_2}, \Gamma(e_1.m(\vec{e_2})) = \tau_3$	$\texttt{method}(\tau_1, m, \vec{\tau_4}, \tau_3)$
			$\vec{\tau_4} = [\text{create } \tau_p \text{ for } \tau_s \text{ in } \vec{\tau_2}]$	$\forall \langle \tau_s, \tau_p \rangle \in \langle \vec{\tau_2}, \vec{\tau_4} \rangle, \texttt{subtype}(\tau_p, \tau_s)$
	New Instance	new $t(\vec{e})$	$t : \tau_1, \Gamma(\vec{e}) = \vec{\tau_2}, \Gamma(\text{new } t(\vec{e})) = \tau_3$	$\texttt{method}(\tau_1, \texttt{"<init>"}, \vec{\tau_4}, \tau_3)$
			$\vec{\tau_4} = [\text{create } \tau_p \text{ for } \tau_s \text{ in } \vec{\tau_2}]$	$\forall \langle \tau_s, \tau_p \rangle \in \langle \vec{\tau_2}, \vec{\tau_4} \rangle, \texttt{subtype}(\tau_p, \tau_s)$
	Instance Of	e instanceof t	$t = \tau_1, \Gamma(e \text{ instanceof } t) = \tau_2$	$\texttt{subtype}(\tau_2, \text{"boolean"})$
	Array Access	$e_1[e_2]$	$\Gamma(e_1) = \tau_1, \Gamma(e_2) = \tau_2$	$\texttt{subtype}(\tau_2, \text{"int"})$

Extracting Constraints

Based on the type variables created from a type element, SnR further extracts constraints from the type element to capture the relations among the type variables. Table 3.3 lists all the types of constraints used in SnR, and Table 3.4 lists the concrete rules to create type variables and constraints for type elements.

Take the class declaration for an example. The rule with the name 'Declaration' in Table 3.4 specifies that for a class declaration c, SnR generates at least two type variables

[1]Vector $\vec{\tau_4}$ is created for method and new instance expression where new type variables τ_p is created for each type variable in $\vec{\tau_2}$; for each pair of original and newly created type variable $\langle \tau_s, \tau_p \rangle$ from the two vectors $\langle \vec{\tau_2}, \vec{\tau_4} \rangle$, generate a `subtype` constraint.

(τ for c and τ_1 for the super class of c), and a list of type variables $\vec{\tau_2}$ ($\vec{\tau_2}$ can be empty) for the interfaces $\vec{t_2}$ of c with each type variable τ_i in $\vec{\tau_2}$ representing the type of one interface. Then SnR creates one constraint `extend`(τ, τ_1) to capture the typing relation between c and t_1, and one `interface`(τ, τ_i) for each implemented interface.

SnR makes extensive use of the `subtype` constraint in order to model implicit type conversions of both primitive and reference types allowed in Java (*e.g.* in assignments or when passing method arguments) and to constrain certain known types in the AST (*e.g.* the conditional in an if statement is a boolean).

An Example. Following the rules outlined in Table 3.4, SnR creates the following constraints over the type variables τ_1, τ_2, τ_3, and τ_4 of the example statement in §3.3.2.

Constraints	Description
`simplename(`τ_1`,"Date")`	τ_1 has the simple name `Date`
`simplename(`τ_2`,"Calendar")`	τ_2 has the simple name `Calendar`
`method(`τ_2`,"getInstance",[],`τ_3`)`	τ_2 has a method named `getInstance` that takes no arguments with the return type of τ_3
`method(`τ_3`,"getTime",[],`τ_4`)`	τ_3 has a method named `getTime` that takes no arguments with the return type of τ_4
`subtype(`τ_1`,`τ_4`)`	τ_1 has a subtype τ_4 because of the assignment

3.3.3 Type Inference: Resolving Ambiguities

From the constraint generation rules laid out in Table 3.4, inner classes, qualified names, and field expressions can bring ambiguities into the process of extracting constraints. For example,

```
java.util.Collections.EMPTY_LIST
```

for the first identifier `java` in this expression, there are four broad categories of possible interpretations,

① `java` is a variable defined locally in the current code snippet, *e.g.*, a local variable, a method parameter, or a class field.
② `java` is a class defined locally in the current code snippet, *e.g.*, an inner class.
③ `java` is the name of a type (*e.g.*, a class or an interface) that is defined externally but not in the current code snippet.
④ `java` is a part of a package name, and the package name is used to form a FQN.

The remaining identifiers are then potentially a mix of packages, classes, inner classes, and field references. It is impossible at the parsing time to determine which of these cases the current code snippet refers to without import statements.

SnR verifies these interpretations in the order they are listed. ① and ② can be verified by examining the code snippet for locally defined variables and types. On the other hand, without import statements, ③ and ④ cannot be verified. SnR overcomes this challenge by leveraging the knowledge base. For the example above, by performing a knowledge base lookup $\Omega_{simplename}$(`"java"`), ③ can be ruled out. Similarly, by performing multiple Ω_{fqn} queries to find the longest matching FQN, we find that ④ `java.util.Collections` is the best possible interpretation to resolve the ambiguity.

Inner Class and Field Constraints. The subsequent unmatched parts could be inner classes or fields. Inner classes can be found by performing additional look-ups to the knowledge base ($\Omega_{simplename}$). The rest of the identifiers are considered to be fields.

In the event that the knowledge base is not complete, *i.e.*, the knowledge base does not have information about the symbol, then SnR may generate incorrect constraints. However, this can be easily addressed by incorporating more libraries into the knowledge base.

3.3.4 Type Inference: Solving Constraints

Given a code snippet s, the set C of the constraints extracted from s by the rules in Table 3.4 can be directly solved by a Datalog solver against the knowledge base, if s does not define or use any generic methods or types.

However, if s uses generics inside, due to the complexities of Java generics and the limited expressiveness of Datalog, we propose a two-step process to solve C. Generally speaking, for s and C, SnR first generates a new, possibly smaller knowledge base from the original knowledge base Ω by replacing generic types in Ω with concrete types, and then uses the small knowledge base as Datalog facts to solve C.

The small knowledge base can be viewed as a *refinement* of Ω with more concrete type information, and thus is referred to as a *refined* knowledge base. The main reason for introducing this refinement step is that Ω only has signatures of generic types and methods and it is not easy to encode the typing rules of Java generics completely in Datalog.

Example

We use the following code snippet as an example to demonstrate how refined knowledge bases are generated.

```java
List<Date> lod = new ArrayList<>();
lod.get(0);
```

The following table shows the constraints extracted from the code snippet above. There are five type variables in total. The purposes of τ_1, τ_2, and τ_3 are for `List`, `Date`, and `ArrayList` respectively; τ_4 represents the type of `List<Date>` via the constraint `paramtype`(τ_1, τ_2, τ_4), while τ_5 is the return type of `lod.get(0)`.

Constraint	Constraint
`simplename`$(\tau_1,$ `"List"`$)$	`paramtype`(τ_1, τ_2, τ_4)
`simplename`$(\tau_2,$ `"Date"`$)$	`subtype`(τ_4, τ_3)
`simplename`$(\tau_3,$ `"ArrayList"`$)$	`method`$(\tau_4,$ `"get"`$,$ `["int"]`$, \tau_5)$

Querying Type Candidates. To instantiate the generic types in the original knowledge base Ω, we need to retrieve all the types from Ω that are relevant to the constraints. Therefore for each type variable τ and each `simplename`, `fqn`, `field`, and `method` constraints on τ, we use the corresponding query functions defined in Table 3.2 to retrieve all possible candidate types for τ. For example, for the `simplename` constraints of τ_1, τ_2, and τ_3 we can use $\Omega_{simplename}$ to retrieve the following type candidates for each type variable (Note that the candidates are pruned for illustration purpose.).

Constraint	Query	Candidates
`simplename`$(\tau_1,$ `"List"`$)$	$\Omega_{simplename}($`"List"`$)$	`java.util.List` `java.awt.List`
`simplename`$(\tau_2,$ `"Date"`$)$	$\Omega_{simplename}($`"Date"`$)$	`java.util.Date` `java.sql.Date`
`simplename`$(\tau_3,$ `"ArrayList"`$)$	$\Omega_{simplename}($`"ArrayList"`$)$	`java.util.ArrayList`

Building Refined Knowledge Bases. Based on the information of the type candidates, we next build the refined knowledge base. We first look into C to find `paramtype` constraints which represent instantiations of generic types in code snippets. It is `paramtype`$(\tau_1,$ $\tau_2, \tau_4)$ in this example. From this constraint, we know that τ_1 should be a generic type, and therefore we remove `java.awt.List` from the candidate set of τ_1 as this class is not generic; τ_2 can be either `java.util.Date` or `java.sql.Date`; therefore, τ_4 can be either of the following two concrete types

- `java.util.List<java.util.Date>`
- `java.util.List<java.sql.Date>`

From the constraint $\texttt{subtype}(\tau_4, \tau_3)$, we need to further instantiate two concrete classes of `java.util.ArrayList` as follows,

- `java.util.ArrayList<java.util.Date>`
- `java.util.ArrayList<java.sql.Date>`

Next, we create a refined knowledge base Ω' by combining these four concrete classes and Ω. The method `T java.util.List.get(int)` in Ω becomes the following two methods in Ω',

- `java.util.Date java.util.List<java.util.Date>.get(int)`
- `java.sql.Date java.util.List<java.sql.Date>.get(int)`

Constraint Solving. In the end, we use a Datalog solver to solve C by using Ω' as the Datalog facts. The benefit of using Ω' is obvious: when the Datalog solver sets τ_2 to either concrete `Date` type, τ_5—the return type of the method call `get(0)`—will have the same type as τ_2, thanks to the specialized `get(int)` methods in Ω'. In contrast, if we use Ω as the Datalog facts, the Datalog solver cannot infer that τ_5 should always be the same as τ_2.

3.3.5 Type Inference: Candidate Prioritization

Given a code snippet s and the set of type variables T extracted from s, the type inference engine in §3.3.4 outputs a set of FQNs for each $\tau \in T$ as well as sets of libraries defining each FQN.

Solution Candidates. Then SnR processes the type inference result of c and outputs a list of solution candidates. Each candidate is a set of triples in the form

$$\{\langle \tau, \mathit{fqn}, \mathit{lib} \rangle | \tau \in T, \mathit{fqn} \text{ is a FQN for } \tau, \mathit{lib} \text{ is a library defining } \mathit{fqn}\}$$

Prioritization Heuristic. To make SnR useful and accurate at repairing code snippets, we design a simple yet effective prioritization heuristic to rank these solution candidates so that the candidates at the top of the ranking list are more likely to be correct than those at the bottom. The general principle of this heuristic is to minimize the number of unique libraries in a candidate. Take Table 3.1 for an example. The third candidate is ranked after the first two because the third one has one more library, *i.e.*, `gwt`. The intuition behind our heuristic is similar to the clustering hypothesis proposed in [53].

Table 3.5: Statistics of the StatType-SO benchmark.

(a) The number of public or protected, classes, fields, and methods for each library in StatType-SO.

Library	Classes	Fields	Methods
Android	2,357	8,943	22,933
JDK	11,881	28,443	105,807
JodaTime	143	166	3,053
GWT	1,518	542	9,288
Hibernate	2,356	1,681	18,749
XStream	628	146	3,855
Total	18,883	39,921	163,685

(b) Top 5 simple class names in the StatType-SO dataset with their respective number of occurrences.

Class Name	Occurrences
Builder	71
EntrySet	40
Type	38
Entry	30
PropertyKeys	29

3.4 Evaluations

We have conducted extensive evaluations of SnR in different aspects to answer the following research questions.

RQ1: How does SnR perform at type inference?
RQ2: How does SnR perform at resolving import statements?
RQ3: Does SnR recommend the correct libraries?
RQ4: How does SnR perform at making code snippets compilable?

RQ1 to RQ4 evaluate the performance of SnR compared to the state-of-the-art type inference tools. Since Coster has been shown to match or outperform prior techniques [3]; thus in this thesis we only compare SnR with Coster. The source code of Coster, along with their model, was obtained from its GitHub repository [55].

We use precision, recall, and F_1 scores to measure the performance of SnR considering only the top candidate, as used by Coster.

$$Precision = \frac{recommendations\ made\ \cap\ relevant}{recommendations\ made}$$

$$Recall = \frac{recommendations\ made\ \cap\ relevant}{recommendations\ requested}$$

$$F_1 = \frac{2\ \times Precision \times Recall}{Precison\ +\ Recall}$$

Dataset. We use an existing dataset referred to as StatType-SO, which was used in [2, 3]. The dataset consists of 267 snippets from six popular libraries. Table 3.5 lists various statistics of StatType-SO. All the public classes, fields, and methods are counted not including inherited fields and methods. This dataset represents how developers use a wide variety of real Java libraries in practice, and evaluations using this dataset demonstrate that our technique is sound for libraries ranging from small to large. This benchmark which consists of the code snippets and the libraries was obtained from the original benchmark authors Phan et al. [2].

Implementation. SnR is a single-threaded Java application and uses MariaDB to serve as the knowledge base. The constraints are solved using the Soufflé [20] Datalog solver. We use the Eclipse Java Compiler to create and traverse ASTs. A replication package is available at `https://doi.org/10.5281/zenodo.5843327`.

Hardware Configuration. All experiments were conducted on an eight-year-old laptop with Intel Core i5-4300m CPU 2.60 GHz and 16GB RAM. The operating system is Linux.

3.4.1 RQ1: How does SnR perform at type inference?

We measured SnR's and Coster's performance in recommending FQNs for API elements (*i.e.*, simple names, field accesses, and method calls as defined in [3]) in StatType-SO. The results are summarized in Table 3.6. SnR significantly outperforms Coster in precision (98.20% vs 66.35%) and recall (79.66% vs 66.35%). Coster often incorrectly recommends the more popular Apache Commons logging library as opposed to `android.util.Log`, despite the fact they have different logging methods. SnR on the other hand is able to achieve 100% precision for three out of the six libraries in the dataset.

Table 3.6: Performance of type inference for API elements.

	Coster		SnR	
	Precision	Recall	Precision	Recall
Android	43.28%	43.28%	100.00%	93.64%
JDK	56.24%	56.24%	97.37%	71.12%
JodaTime	57.14%	57.14%	100.00%	89.47%
GWT	90.75%	90.75%	96.68%	75.84%
Hibernate	90.38%	90.38%	99.32%	94.81%
XStream	88.41%	88.41%	100.00%	100.00%
Total	66.35%	66.35%	98.20%	79.66%

Table 3.7: Performance of type inference for type elements.

	Total	Analyzed	Correct	Precision	Recall
Android	1,690	1,452	1,308	90.08%	77.40%
JDK	12,450	10,245	9,166	89.47%	73.62%
JodaTime	1,283	1,051	1,013	96.38%	78.96%
GWT	2,273	1,951	1,679	86.06%	73.87%
Hibernate	1,583	1,496	1,407	94.05%	88.88%
XStream	864	864	804	93.06%	93.06%
Total	20,143	17,059	15,377	90.14%	76.34%

To further understand the performance of SnR, we applied SnR to infer FQNs for all type elements (defined in Definition 3.3.1) in StatType-SO, which is a much more difficult task than inferring FQNs for API elements because type elements are a large super set of API elements. Note that Coster is not capable of doing this task.

To compute the ground truth, for each code snippet, we provided the proper libraries to Eclipse and used Eclipse to find and compute types for the type elements in the dataset. In the end, Eclipse found 20,143 type elements along with their types in total.

Table 3.7 shows the performance of SnR on this task. Among those type elements, SnR analyzed 17,059 ones and 15,377 were correctly inferred, achieving a precision of 90.14% and recall of 76.34%. This high precision and recall for all type elements further demonstrates the advantages of SnR over the state of the art, which also enables SnR to accurately, and reliably repair incomplete code snippets.

3.4.2 RQ2: How does SnR perform at resolving import statements?

We use SnR and Coster to resolve import statements for all code snippets. Because Coster was not explicitly designed to support catch and annotation expressions, we automatically completed 56 of those import statements for Coster.

Table 3.8 summarizes the results. Overall, SnR correctly resolved 91.0% of the import statements, whereas Coster could only resolve 36% though 56 import statements were resolved by us.

Different from Table 3.8 which shows the information on completed import statements,

Table 3.8: Performance of inferring import statements.

	Total Imports	SnR Completed	Coster Completed
Android	220	205 (93.2%)	18 (8.2%)
JDK	332	294 (88.6%)	174 (52.4%)
JodaTime	134	118 (88.1%)	35 (26.1%)
GWT	301	265 (88.0%)	90 (29.9%)
Hibernate	159	149 (93.7%)	101 (63.5%)
XStream	152	150 (98.7%)	49 (32.2%)
Total	1,298	1,181 (91.0%)	467 (36.0%)

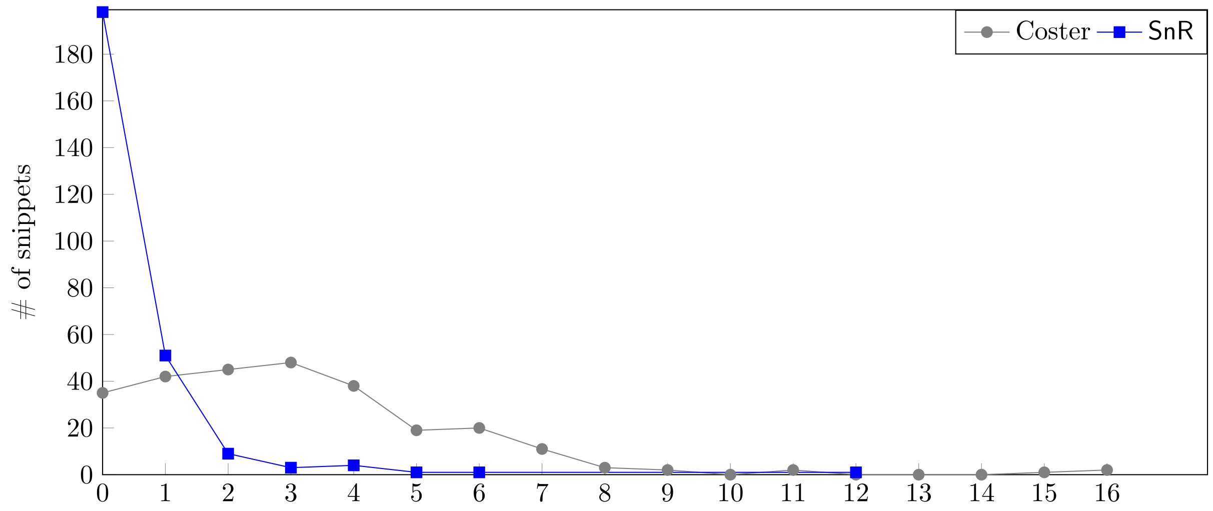


Figure 3.5: The distribution of code snippets *w.r.t.* number of missing import statements after repair using SnR and Coster. Points with $y = 0$ are not plotted.

Figure 3.5 shows the information on missing import statements. The x-axis is the number of missing import statements ranging from 0, and the y-axis is the number of code snippets with the `x` number of missing imports after being repaired by either SnR or Coster. SnR can completely repair 198 code snippets without missing import statements, compared to only 35 by Coster; for SnR, most of the rest code snippets have one or two missing imports, while for Coster, most of the rest have one to seven missing imports. This figure demonstrates that SnR is able to resolve more import statements accurately than Coster, and thus can potentially save more developers' time.

Real-world example. Recently, Coster has been released as an Eclipse plugin [56] for finding FQNs in code snippets. The author produced a demonstration video [57] illustrating the new integration with Eclipse to fix import statements. In the video, the authors attempted to repair the code snippet from a Stack Overflow post [58] for which Coster failed to create import statements for `DateTimeZone` and `DateTimeFormat`. We applied SnR on the same code snippet, and SnR precisely resolved *all* import statements.

3.4.3 RQ3: Does SnR recommend the correct libraries?

We evaluated the accuracy of SnR's library recommendation. We manually examined each code snippet in the dataset to find all the dependent libraries, not just the six used in the previous evaluations. There were 33 unique libraries in total.

We compared SnR against a naive ($\text{SnR}_{\text{Naive}}$) approach where libraries are sorted alphabetically and taken greedily until all the missing libraries are satisfied, to validate whether our candidate prioritization heuristic detailed in §3.3.5 is effective. We also compared against Coster which recommends all libraries containing the inferred FQNs. For each code snippet, the recommendation result is classified into one of the following categories.

Same if the tool recommends the exact expected libraries.
Different if the tool recommends one or more alternatives for some expected libraries.
Extra if the tool recommends a superset of the expected libraries.
Missing if the tool recommends a subset of the expected libraries.
None if the tool incorrectly recommends no libraries.

Table 3.9 lists the classification result. SnR correctly recommended the exact libraries for 183 code snippets, compared to 118 for $\text{SnR}_{\text{Naive}}$ and 34 for Coster. SnR also achieved the highest precision and recall, and thus we concluded that SnR performed the best among the three tools, which further demonstrates that our candidate prioritization strategy in §3.3.5 is effective in improving the accuracy of type inference.

Table 3.9: SnR library recommendation compared to a naive candidate prioritization $\text{SnR}_{\text{Naive}}$, and Coster.

	Same	Different	Extra	Missing	None	Precision	Recall	F_1
SnR	183	62	11	4	7	0.82	0.83	0.82
$\text{SnR}_{\text{Naive}}$	118	75	64	3	7	0.68	0.81	0.72
Coster	34	36	129	8	60	0.47	0.68	0.53

3.4.4 RQ4: How does SnR perform at making code snippets compilable?

Efficacy

To evaluate the efficacy of SnR at automatically making code snippets compilable, we compiled the repaired code snippets and recorded the remaining errors. SnR achieved an average of 73.8% where 197 out of 267 snippets were compilable after the repair. Coster on the other hand could make only 9.0% (24 out of 267) of snippets compilable. Our high-precision type inference technique allows for higher-quality repair and results in a larger number of compilable code snippets thus allowing for more information to be recovered from each code snippet.

Efficiency

SnR is efficient enough to be used in practice. Averaged over five runs, SnR's repair process finished in an average of 11.7 seconds for each snippet and half the snippets finished within 8.4 seconds. As seen in Figure 3.6, the time SnR takes to repair a snippet increases as the number of imports in a snippet increases. But even for a very complex code snippet, the slowest one finished in 82.2 seconds on an eight-year-old laptop.

Coster can finish repairing a code snippet very fast in seconds, despite its low efficacy in repairing code snippets for compilation. On the other hand, Coster is based on machine learning, and requires training a model, which can take days to finish.

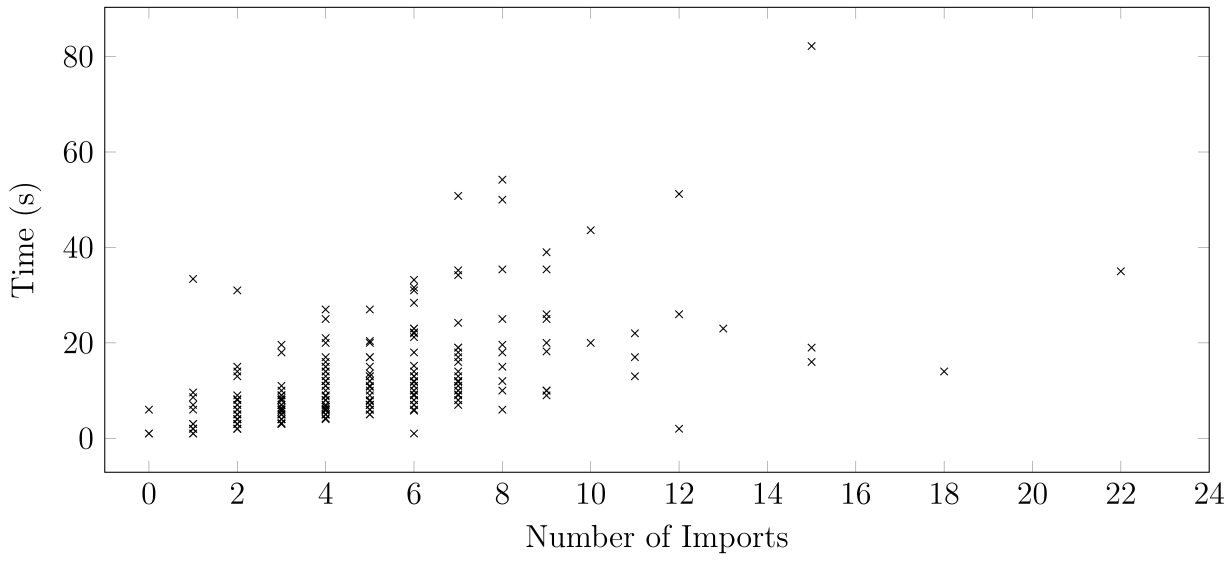


Figure 3.6: The time it takes to repair a snippet with X number of imports. Each $\times$ represents a snippet.

3.5 Discussion

In this section, we will discuss potential applications of our work (§3.5.1), along with the limitations of our technique (§3.5.2) and potential threats to validity (§3.5.3).

3.5.1 Application

SnR has immediate applications for software engineering.

IDE Improvement. The constraint-based technique used in SnR can be readily used by existing IDEs to provide accurate code completion suggestions. For example, given the code snippet shown in Figure 3.1, currently, Eclipse does not properly leverage the relation between `Date` and the other APIs, and thus may incorrectly rank `java.sql.Date` before `java.util.Date`. With the help of SnR, Eclipse can precisely recommend importing `java.util.Date`. Another salient application of SnR is to automatically import dependencies for pasted code snippets. Coster has provided an Eclipse plugin of a similar purpose [56]. As demonstrated in §3.4, SnR outperforms Coster and can effectively improve the performance of such an IDE feature. As code snippets are generally small and developers infrequently copy large chunks of code from Stack Overflow, thus SnR's inference time is sufficient for real world use.

Dependency Repair. Dependency-related issues account for a large number of build failures at Google [59]. The state-of-the-art tool for fixing dependency issues is DeepDelta, which leverages a deep learning model to learn how developers fix such issues in the past, and apply the model for new build failures. SnR can complement DeepDelta, as SnR makes full use of type systems built in programming languages, overlooked by DeepDelta. With SnR, failed compilation due to missing libraries can be automatically repaired by running the inference and adding the suggested libraries.

Stack Overflow Study. SnR can increase the precision and scope of analysis on online code snippets [60, 61, 62, 63, 64]. It can be used to fix the incomplete code snippets, and the followed analyses can take advantage of the compilable snippets which can offer more semantic information about the snippets; to provide better metrics on what makes a good code snippet [60]; to aid training of algorithms that use Stack Overflow snippets [61]; to improve existing IDE Stack Overflow code recommendation tools [62].

3.5.2 Limitation

Certain code snippets reference classes not in the knowledge base and thus cannot be inferred *e.g.* from a class written in a tutorial. This limitation impacts the efficacy of both SnR and previous solutions alike. We address this limitation by providing partial solutions by ignoring the type variables without candidates which may introduce inaccuracies. However, as can be seen from RQ1-4, SnR still outperforms the state-of-the-art tool.

3.5.3 Threats to Validity

Internal. We did our best to minimize potential internal validity issues. Our SnR implementation may contain bugs leading to incorrect repair. We mitigated this by reviewing the instances where SnR is unable to perform repair steps. To ensure the integrity of our evaluations, we externalized the evaluation scripts to ensure both SnR and Coster are evaluated in the same manner. For Coster, we pulled the code and model from their public GitHub repository and followed their instruction to set up their tool. These mitigations ensure we're presenting a fair comparison for SnR.

External. In terms of external validity, there is a risk our work may not generalize to other programming languages. The constraints are fairly universal and can apply to other object-oriented languages. Most languages have fields and methods. Generics are common in other typed languages, *e.g.* C#, Rust, Swift, TypeScript. The constraint solving stage is language agnostic. Thus our technique does not rely on Java specific constructs and can be adapted for other programming languages.

3.6 Related Work

This section discusses prior work in type inference, partial program analysis, Stack Overflow snippet analysis, and automated program repair.

Type Inference. The area of type inference has a rich history in programming languages, especially object-oriented languages [65, 66, 67, 68, 69]. Constraints have also been used for type analysis [70, 67]. Our work departs from existing works in this area by working with incomplete code and leveraging a knowledge base.

Type inference on incomplete code has seen some recent interests [1, 2, 3]. Our work is similar to some of the earlier works from Subramanian et al. [1] in the use of constraints but differs in some key ways such as (1) generic type handling, (2) method of solving, and (3) selection of multiple compatible candidates. Their tool Baker relies on simple constraints and does not provide library recommendations. More recent works surpassed Baker's performance using statistical models to improve inference accuracy [2, 3]. These models are trained using a large set of existing [71] or collected popular GitHub projects and are evaluated using hand-collected Stack Overflow posts including StatType-SO. Saifullah et al. [3] improved upon the model by Phan et al. [2] by leveraging local and global context. Our work improves upon the state of the art and complements the existing techniques. Future techniques can incorporate the accuracy of constraint-based techniques with the performance of statistics-based ones.

Deep learning models [72, 73] have been used to conduct type inference on dynamically typed languages. Unlike SnR, these techniques often use additional sources of information such as comments, and identifier names to conduct their inference.

Partial Program Analysis. RecoDoc [74] is a tool for analyzing partial programs which are subsets of the program source files. Code snippets on the other hand can be considered to be subsets of partial programs. Thus, the techniques for analyzing partial programs are insufficient for analyzing code snippets.

Stack Overflow Snippet. Past research has leveraged Stack Overflow snippets to build better development tools [61, 62], to study usages in open source projects [63, 75, 64, 76], and more [77, 53, 2, 3]. Recent work has used heuristics-based approaches to automatically synthesize compilable code snippets [53, 78]. Our work focuses on type inference and greatly improves upon precision compared to the existing state-of-the-art solution.

Automated Repair. In the area of automated repair, prior research has attempted to address compiler errors using neural networks [59], semantic errors leveraging test cases [79], or specifications (pre- and postconditions) [80]. Our work focuses on code snippets which are incomplete code, without tests or specifications in most cases. Our technique needs to be more flexible and cannot rely on having test cases or specifications.

3.7 Chapter Conclusion

In this chapter, we proposed SnR, a novel, effective, constraint-based technique to automatically infer missing import statements and dependent libraries for Java code snippets. Given an incomplete snippet, SnR first automatically gathers constraints from the snippet, then solves these constraints by querying a knowledge base built from a large collection of Java libraries, and finally transforms the solutions to the constraints to import statements and dependency libraries. Our comprehensive evaluation of SnR on the StatType-SO benchmark consisting of 267 snippets demonstrates that SnR significantly outperforms the state of the art: SnR completed 91.0% of the import statements, and compiled 73.8% of the snippets. Even for the fail-to-repair snippets, our best-effort repair left most of them with only one missing import statement. The high inference precision of SnR opens up new opportunities for boosting developers' productivity with code snippets.

Chapter 4

Evaluating the True Performance of LLMs for Java Code Snippets

After the introduction of SnR, advances in machine learning have enabled LLMs to achieve strong performance in type inference for Java code snippets. This chapter investigates the performance of LLMs in greater depth and demonstrates that existing evaluations may overestimate their true capabilities, potentially due to data leakage.

4.1 Introduction

Current approaches to type inference for Java code snippets generally fall into two categories: constraint-based and machine learning (ML)-based techniques. Constraint-based methods build a knowledge base of libraries and the types contained within [4, 1]. SnR [4] leverages this knowledge base and uses constraints extracted from the code snippet to identify the precise types being used. In contrast, ML-based approaches [2, 3, 5, 7] learn from prior examples of type usage in existing code to perform type inference.

ML-based techniques often produce incorrect inferences, due to their limited understanding of code structure and the rules governing Java's type system. Recent advancements in LLMs, such as OpenAI's proprietary GPT family and Meta's open-weight Llama models, offer new possibilities for overcoming these limitations. By leveraging training on diverse and extensive datasets, LLMs have shown potential in performing various software engineering tasks [81]. Prior studies [5, 7, 8] suggest that LLMs achieved performance comparable to that of the state-of-the-art, SnR.

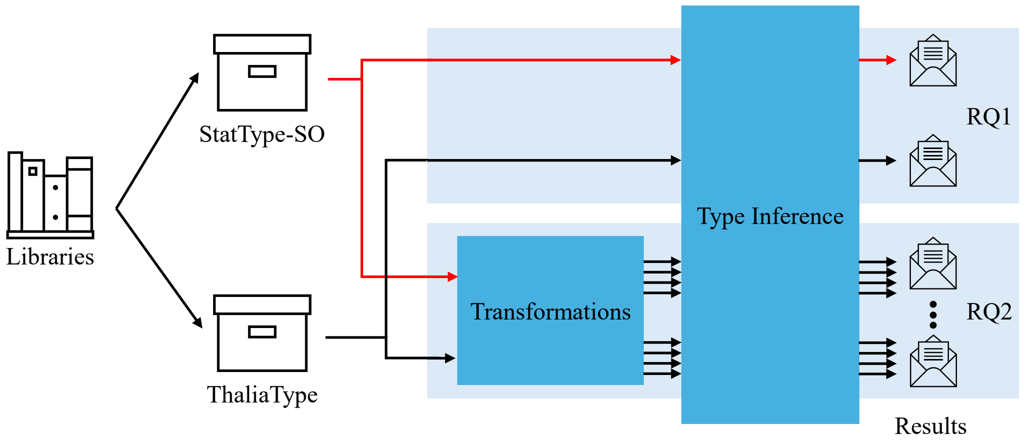


Figure 4.1: The general workflow. The libraries used to collect code snippets in StatType-SO were also used to generate code snippets in ThaliaType. These code snippets are used in RQ1 and their transformed versions are used in RQ2 to evaluate various type inference techniques. Their results are compared to evaluate the true type inference capabilities of LLMs for Java code snippets. The red color represents the potential for data leakage.

However, the evaluation of the prior techniques [5, 7, 8] is potentially affected by *data leakage* [82] where the training dataset includes the benchmark dataset, and data leakage allows LLMs to regurgitate the training data rather than conducting type inference. The benchmark suite, StatType-SO, used for evaluating type inference techniques has been fully, publicly available on GitHub since 2017 [83]. As shown in Table 4.1, recent state-of-the-art LLMs have knowledge cutoff dates in late 2023, creating the potential for leakage. Thus it remains uncertain whether the LLMs' strong performance stems from their ability to understand the semantics of the code snippets or merely from retrieving the ground truth from their training data. Since the exact training data for LLMs are kept confidential, detecting such leakage is difficult [84, 85]. While recent work has called attention to this data leakage problem for software engineering research [86, 50], *no study has thoroughly, empirically evaluated LLMs' performance on type inference on code snippets.*

In this chapter, we thoroughly evaluate LLMs' type inference performance on Java code snippets to better understand the strengths and limitations of LLMs on type inference, the impact of data leakage, and uncover potential limitations of LLMs.

First, we investigated StarCoder2, an open-weight LLM trained on a publicly available training dataset The Stack v2. From exploring The Stack v2, all code snippets from StatType-SO were found, meaning that StarCoder2 was trained using code snippets from StatType-SO. Thus StarCoder2 can potentially recall the code snippets and their expected import statements from training rather than performing type inference. StarCoder2's performance is investigated alongside Llama and GPT models to glean further insight into the behaviors of LLMs. We organized our inquiry into the following two research questions.

RQ1: How well do LLMs perform type inference on unseen code snippets?

To address data leakage, we generated a new dataset named ThaliaType using Thalia, a program synthesis technique originally designed for testing static type systems. ThaliaType comprises 300 code snippets guaranteed to be unseen by the evaluated LLMs, thereby avoiding data leakage. Furthermore, we used the same set of libraries as used in StatType-SO to generate ThaliaType, (1) to enable a direct comparison of type inference performance between the two datasets, and (2) to ensure the evaluated LLMs were consistently and sufficiently trained on the same set of types used in both datasets. By evaluating LLMs on ThaliaType and contrasting those results with those from StatType-SO, we can better understand the impact of data leakage on performance.

RQ2: To what extent do LLMs understand the execution semantics of code snippets?

To investigate whether LLMs truly understand the semantics of code snippets during type inference, we designed semantic-preserving code transformations to create syntactically different versions of the input code (§4.6). These transformations were intended to hinder LLMs' reliance on memorized training data by making direct recall more challenging. By observing how LLMs' performance changes on these semantically equivalent but syntactically distinct versions, we can gain insights into their ability to grasp the underlying semantic meaning of the code.

Following the recommendation from prior software engineering research [50], we evaluate LLM's performance using three open-weight models StarCoder2, Llama3.1:8b, and Llama3.1:70b, along with state-of-the-art closed-source models GPT-4o and GPT-4o-mini (see Table 4.1b). Among these, StarCoder2 serves as a baseline model with confirmed data leakage (see §4.4.2). For additional comparison, we include SnR [4], a state-of-the-art constraint-based type inference technique, as a non-LLM baseline. This evaluation aims to provide deeper insights into the potential limitations of using LLMs and to guide future research in evaluating LLMs' performance on other software engineering tasks.

In RQ1, we found LLMs are highly likely to be affected by the data leakage issue and are prone to making incorrect inferences on code snippets from ThaliaType. All tested LLMs exhibited a similar decline in performance as StarCoder2:15b which experienced a 66.8% decrease in F1-score. For example, Llama3.1:8b and GPT-4o showed comparable declines of 67.2% and 48.5% respectively. On both StatType-SO and ThaliaType,

Table 4.1: Overview of the datasets and language models used to evaluate LLM performance. The early publication of StatType-SO, combined with the late knowledge cutoff dates of the models, increases the likelihood of data leakage.

(a) The datasets used for evaluation.

Dataset	Publication Year	#Code Snippets
StatType-SO	2017 [83]	267
ThaliaType	Not Applicable	300

(b) Evaluated models with their respective version identifiers, knowledge cutoff dates, and whether StatType-SO was present in their training data. The question mark denotes that StatType-SO's presence cannot be confirmed.

Model	Version	Knowledge Cutoff	Data Leakage
StarCoder2:15b	instruct-v0.1-q4_0	September 2023 [26]	✓
GPT-4o	2024-08-06	October 2023 [87]	?
GPT-4o-mini	2024-07-18	October 2023 [87]	?
Llama3.1:8b	instruct_q4_0	December 2023 [25]	?
Llama3.1:70b	instruct_q4_0	December 2023 [25]	?

StarCoder2:15b outperformed Llama3.1:8b but was outperformed by Llama3.1:70b, GPT-4o-mini, and GPT-4o. In contrast, SnR, which is not affected by data leakage, obtained a precision of 84.15% and a recall of 84.43% on ThaliaType. The consistent drop in LLMs' performance between the baseline StarCoder2:15b and other models suggests that all tested LLMs are potentially affected by data leakage. In addition, although StarCoder2:15b was trained on StatType-SO snippets, it only achieved an F1-score of 81.67%, indicating that even in the presence of data leakage, models do not perfectly recall the training data.

In RQ2, LLMs showed robustness to individual transformations on both benchmark suites, and also to the combined transformation on ThaliaType code snippets. StarCoder2:15b showed only a 3.0% decrease in F1-score in the worse case, while Llama3.1:8b experienced a 7.9% drop. *However*, combined transformations on StatType-SO consistently led to performance degradation across all models, including a 16.8% reduction for StarCoder2 and up to 30.0% for Llama3.1:8b. Notably, these same combined transformations did not consistently degrade performance on ThaliaType. In fact, StarCoder2:15b experienced a slight improvement in F1-score. One possible explanation is the presence of data leakage, which may allow models to recall answers from training. When transformations introduce enough variation to prevent direct recall, the models are forced to rely on semantic reasoning. Under these conditions, their performance on StatType-SO aligned more closely with that on truly unseen data.

Therefore, future evaluations of LLM-based techniques should also evaluate generalizability, defined as the ability for LLM-based techniques to maintain consistent performance across unseen code snippets, by incorporating both transformed code snippets and ThaliaType, rather than relying solely on StatType-SO.

Contribution. We make the following major contributions.

- We introduce ThaliaType, a new dataset constructed through a novel adaptation of Thalia for rigorously evaluating the type inference performance of LLMs while mitigating potential bias from data leakage. Using ThaliaType, we demonstrate that prior evaluations based on StatType-SO are likely affected by data leakage, leading to inflated and potentially misleading performance results.

- We evaluate whether LLMs understand the execution semantics of code snippets by applying transformations that preserve the semantics and comparing their effects. Our results on StatType-SO and ThaliaType show that LLMs are generally able to maintain performance across individual transformations. However, complex transformations on StatType-SO result in disproportionately large performance drops compared to both their effects on ThaliaType and the individual transformations. This lack of generalizability can be caused by data leakage where models learned the correct answer from training.

- To enable replication, reproduction, and further research on LLMs for code-related tasks, we have made our dataset and code publicly available at `https://github.com/uw-pluverse/thalia-type`.

4.2 LLM-Based Type Inference

Recent LLM-based approaches have recently achieved state-of-the-art performance on type inference on the StatType-SO [5, 7, 8] dataset. Existing approaches typically employ a simple but effective prompt to guide LLMs in generating FQNs for types in the code snippet [5, 7, 8]. Notably, Kabir et al. [7] extended the single-prompt approach by incorporating a secondary prompt to address compilation errors in the code snippet generated by the first prompt. This iterative process achieved a precision of 99.5% on the StatType-SO

System	You are a helpful programming assistant.
User	Add import statements to the following Java code. Do not use wildcard imports. Include only the necessary import statements. Do not import nonexistent types. Please note that you need to pay close attention and your response should be specific and accurate. Avoid repetition. Reply with the import statements only. `{input_code}`

(a) Example prompt used to infer types on code snippets with LLMs. The `{input_code}` is replaced by the code snippet.

~~~
```java
import java.util.List;
```
~~~

(b) Example response from LLMs using the example prompt.

Figure 4.2: Example prompt used to infer types on code snippets with LLMs, along with an example response. The placeholder `{input_code}` is substituted with the given code snippet.

dataset. In contrast, our work focuses on the issue of *data leakage* in type inference rather than repairing code snippets. Data leakage is a fundamental problem that compromises the validity of results, regardless of whether a single-prompt or iterative approach is used. To ensure consistency and comparability with prior research, we adopted a single-prompt approach for type inference. Following established practices [5, 7], we developed a straightforward but highly effective prompt for LLM-based type inference.

Prompt Template. Our prompt, shown in Figure 4.2, is designed to guide the LLM in inferring the type information in Java code snippets. It includes two key components: (1) a system message that sets the LLM's role as a helpful assistant, and (2) a user instruction explicitly requesting the addition of import statements for Java code. The placeholder `{input_code}` is replaced with the actual code snippet during inference. Figure 4.2a presents the exact wording of the prompt, while Figure 4.2b demonstrates the corresponding output generated by the LLM for an example code snippet. Despite its simplicity, our prompt achieves high accuracy and aligns with best practices for LLM prompt engineering [88]. Notably, our result using GPT-4o on the StatType-SO dataset represents a 7% improvement over the prior results [7], demonstrating the robustness of our prompt while maintaining a similar level of complexity.

4.3 Datasets

We utilize two datasets StatType-SO and ThaliaType for evaluating LLMs' type inference performance on code snippets.

4.3.1 StatType-SO

The StatType-SO dataset is prevalent for testing type inference tools [2, 3, 4, 89] including SnR in §3. It is constructed from real-world, manually repaired Java code snippets from Stack Overflow. StatType-SO contains 267 code snippets from six popular Java libraries (Android, GWT, Hibernate, JDK, JodaTime, XStream). The snippets were extracted from 50 randomly selected Stack Overflow posts for each library. To prepare the dataset, all parsing errors in the code snippets were first corrected. Next, the missing import statements were manually inferred and added to the code snippet, serving as the ground truth for evaluation. Figure 4.3 shows an example from StatType-SO. The code snippets from StatType-SO are generally short and make a small number of calls to the libraries and assign the intermediate results to variables.

While StatType-SO is a widely adopted dataset for evaluating type inference techniques, it is important to acknowledge the potential for data leakage. Given its long-standing availability since 2017 at GitHub, StatType-SO may have been inadvertently incorporated into the training data of some LLMs, potentially leading to artificially inflated performance scores.

4.3.2 ThaliaType: A New Dataset

In this thesis, we introduce ThaliaType, a new dataset specifically designed to *rigorously evaluate* the type inference capabilities of LLMs while *mitigating the potential biases* introduced by data leakage. ThaliaType is generated using Thalia, a tool originally developed for testing compilers' type checkers. Recognizing its potential for LLM evaluation, we repurposed Thalia [90] to generate a diverse set of syntactically valid, well-typed, and type-intensive Java programs. Given a set of types, along with the fields and methods defined for each type, Thalia generates programs that utilize a subset of these components. Each program consists of a single class with a main method containing variable declarations, field accesses, and method calls.

Thalia's ability to generate syntactically correct and well-typed programs, combined with their type-intensive nature, aligns perfectly with the requirements for rigorous type

```java
package jodatime;

import org.joda.time.Chronology;
import org.joda.time.DateTime;
import org.joda.time.DateTimeZone;
import org.joda.time.chrono.GJChronology;

//ID = 2182921
public class JodaTime05 {
 public static void main(String[] args) {
     DateTimeZone zone = DateTimeZone.forID("Europe/London");
     Chronology coptic = GJChronology.getInstance(zone);

     DateTime dt = new DateTime(coptic);
     DateTime minusOneDay = dt.minusDays(1);

     System.out.println(minusOneDay );
 }
}
```

Figure 4.3: Example code snippet from StatType-SO using the JodaTime and JDK libraries. Excessive newlines have been removed for clarity of presentation. These code snippets are generally short. The import statements serve as the ground truth and are removed before the code snippet is used for evaluating type inference.

inference evaluation. Thalia ensures that the programs are both valid and well-typed, eliminating the need for manual code cleanup or ground truth labeling. Furthermore, the type-intensive nature of these programs presents a meaningful challenge for type inference, while the inclusion of field accesses and method calls provides useful context for LLMs during inference.

To create ThaliaType, we used Thalia to generate 300 code snippets, with the types, fields, and methods drawn from the six libraries used in StatType-SO. The general process is outlined in Algorithm 1. For each library, 50 code snippets are generated. By selecting the same libraries, we ensured that the types in the generated code snippets were familiar to the LLMs, eliminating performance differences based on varying familiarity with the libraries. Each snippet is based on a single library, following the convention in StatType-SO.

In Figure 4.4, an example of a code snippet generated by ThaliaType is shown. Each code snippet consists of a `Main` class and a `test` method. In this code snippet, the code

```java
package src.toady;

import org.joda.time.chrono.ZonedChronology;
import org.joda.time.Chronology;
import org.joda.time.LocalTime;
import org.joda.time.DateMidnight;

class Main {
  static public final <K, I extends ZonedChronology, X> void test()
      throws Exception {
    long elf = (long)-58;
    Chronology pantsuits = new LocalTime(elf).getChronology();
    DateMidnight fitness = DateMidnight.now(pantsuits);
    int liner = fitness.getDayOfYear();
  }
}

interface Function0<R> {
  public R apply();
}

interface Function1<A1, R> {
  public R apply(A1 a1);
}

interface Function2<A1, A2, R> {
  public R apply(A1 a1, A2 a2);
}

interface Function3<A1, A2, A3, R> {
  public R apply(A1 a1, A2 a2, A3 a3);
}
```

Figure 4.4: Formatted code snippet generated by Thalia using the JodaTime and JDK libraries. The variable names are randomly generated. The function interfaces were generated by Thalia but are unused in ThaliaType code snippets.

Algorithm 1: Generate ThaliaType from libraries. Similar to StatType-SO, each code snippet primarily uses a single API.

```
1 def generate(libraries):
      Input: libraries. Set of jar libraries, i.e., Android, GWT, Hibernate, JDK, JodaTime, and
             XStream.
      Input: ExtractAPIs(library). Function that extracts public types, fields, and methods from
             a library.
      Input: Thalia(apis, libs, N): Function that generates N code snippets that uses the given
             apis and compilable with libs.
      Output: code_snippets: List of generated code snippets.
2     code_snippets ← [ ]
3     for library in libraries:
4         apis ← ExtractAPIs(library)                  # Extract public classes, fields, and methods.
5         libs ← {JDK, library, ...dependencies}       # Libraries required for compilation.
6         snippets ← Thalia(apis, libs, 50)            # Generate 50 code snippets.
7         code_snippets.append(snippets)
8     return code_snippets
```

creates a new `LocalTime` from a given time represented by a `long` value of -58. The chronology from this `LocalTime`, which in JodaTime is `ISOChronology`, is then used to construct the current `DateMidnight`. Finally, `getDayOfYear()` method is called to retrieve the ordinal day number of the year.

The code snippets in ThaliaType are similar to the code snippets from StatType-SO. Figure 4.5 compares the statistics of the two benchmark suits. The code snippets in both datasets are similar in terms of lines of code and the number of assignments. While Thalia-generated programs are type-intensive, ThaliaType includes fewer method calls but utilizes more types (via import statements) in each snippet.

Our experiments show that SnR achieved comparable performance on both ThaliaType and StatType-SO (§4.5.2), demonstrating that ThaliaType provides sufficient semantic information for the type inference task. To the best of our knowledge, ThaliaType is the first dataset designed for evaluating LLMs on code snippet type inference while addressing data leakage. Future research can leverage our replication package to generate additional code snippets using Thalia using their desired libraries to evaluate newer state-of-the-art LLMs.

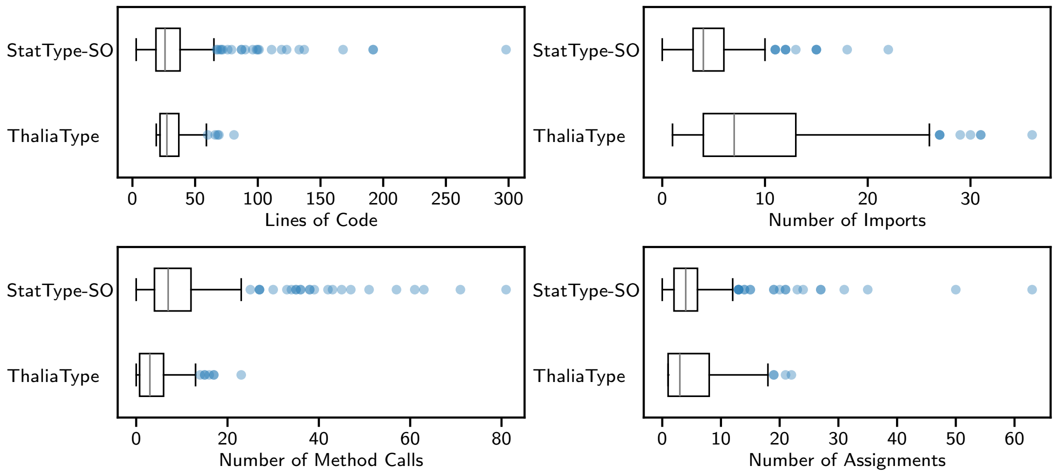


Figure 4.5: Box plots comparing the features found in StatType-SO and ThaliaType.

4.4 Data Sources for LLM Training

LLMs require vast amounts of data as input to train the models.

4.4.1 Proprietary Datasets: GPT and Llama

Both the GPT [24] and Llama [25] families of models are trained on large-scale, proprietary datasets, with details about the training data being only partially disclosed.

OpenAI, the creator of GPT, has stated that the models are trained on a mixture of publicly available data (such as internet text) and data licensed from third-party providers. However, the exact composition of the dataset remains undisclosed due to the proprietary nature of the model, and access to the model is limited to an API.

Similarly, while Llama models are open-weight and thus freely available for download and use, the data used to train them is not publicly released. Meta discloses that Llama is trained on publicly available data, primarily sourced from the internet, but the specifics are not provided. This lack of transparency makes it difficult to assess the risk of data leakage in both models.

Algorithm 2: Finding StatType-SO code snippets in The Stack v2 based on file names.

```
def getRepos(dataset, filenames):
    Input  : dataset. The Stack v2 dataset which contain a list of files and their attributes.
    Input  : filenames. The list of file names for code snippets in the StatType-SO dataset.
    Output: Repositories and authors that are potentially sources of StatType-SO code snippets
            in the dataset are returned.
    repo_counts ← dict()                              # Initialize repository frequency dictionary.
    author_counts ← dict()                            # Initialize author frequency dictionary.
    for data in dataset:
        if basename(data.path()) in filenames:        # Match the file in the dataset using the file name.
            if data.repo() not in repo_counts:
                repo_counts [data.repo()] = 0
            repo_counts [data.repo()] += 1            # Increment count for repo.
            if data.author() not in author_counts:
                author_counts [data.author()] = 0
            author_counts [data.author()] += 1        # Increment count for author.
    return repo_counts, author_counts
```

4.4.2 The Stack v2: StarCoder2

StarCoder2 [26] on the other hand is both open-weight and also trained on an open dataset The Stack v2 that is available to download. The Stack v2 contains 67.5TB of data from 658 different languages. This means that the dataset used to train the model can be examined to determine the extent of the data leakage. Since all our other models are instruction-tuned, meaning they are trained to better follow user instructions, we also used an instruction-tuned StarCoder2 to ensure consistency [27].

Examining The Stack v2. To examine the content of The Stack v2, the meta-data of the files are examined. By matching the file name of the file names in the stack using the process in Algorithm 2, repositories that contained StatType-SO code snippets were extracted. From the file name matching process, there were 26 repositories and three users with five or more matching file names, which served as a starting point for our manual investigation. The manual analysis showed that six repositories were found to have contained StatType-SO code snippets. These six repositories are listed in Table 4.2 and contain all code snippets from the StatType-SO dataset. `mrthlinh/TypeResolution_Oracle` is only missing some of the code snippets that use the `Android` library. `pdhung3012/TypeResolution_Oracle` contains the complete StatType-SO benchmark-suite. All of the repositories found were committed in 2017, far before the knowledge cutoff dates of all LLMs tested. This makes it likely for GPT and Llama to have

Table 4.2: Repositories found in The Stack v2, the number of code snippets, and the last commit date of the repository.

Repository	# of code snippets	Last Commit Date
miketran238/AndroidOracle	50	2017-02-14
miketran238/JodatimeOracle	50	2017-02-12
mrthlinh/Oracle-GWT	50	2017-02-13
mrthlinh/Oracle-Hibernate	50	2017-02-13
mrthlinh/TypeResolution_Oracle	219	2017-02-12
pdhung3012/TypeResolution_Oracle	267	2017-06-30

been trained on some if not all the code snippets in the StatType-SO benchmark suite.

4.5 RQ1: How well do LLMs perform type inference on unseen code snippets?

Given the strong possibility of data leakage, we sought to rigorously assess the true type inference capabilities of LLMs. To do this, RQ1 investigates their performance on two datasets, the original StatType-SO benchmark suite (which may have been seen during training), and the unseen code snippets in ThaliaType. This comparison allows us to isolate the impact of potential training overlap and better understand how well these models generalize to genuinely novel code.

4.5.1 Method

Code snippets from StatType-SO and ThaliaType are given to LLMs using the prompt pattern in Figure 4.2 and also our baseline SnR one at a time *without* import statements. To check the output, import statements in LLMs' response are extracted and compared against the original import statements in the code snippets. To ensure reproducibility, all the inferences are performed with a fixed seed (set to one) and a temperature of zero. GPT-4o-mini and GPT-4o are accessed using the OpenAI API [91], while StarCoder2:15b, Llama3.1:8b and Llama3.1:70b are accessed through a self-hosted Ollama API [92] running on an A100 GPU. The total cost of using the GPT models for all experiments conducted in this chapter was $29.69, based on OpenAI's pricing as of early 2025.

The precision, recall, and F1-scores are computed the same as in the previous chapter to evaluate the performance of type inference. Precision measures the proportion of correctly inferred FQNs among all inferred FQNs, while recall measures the proportion of expected FQNs that are correctly inferred.

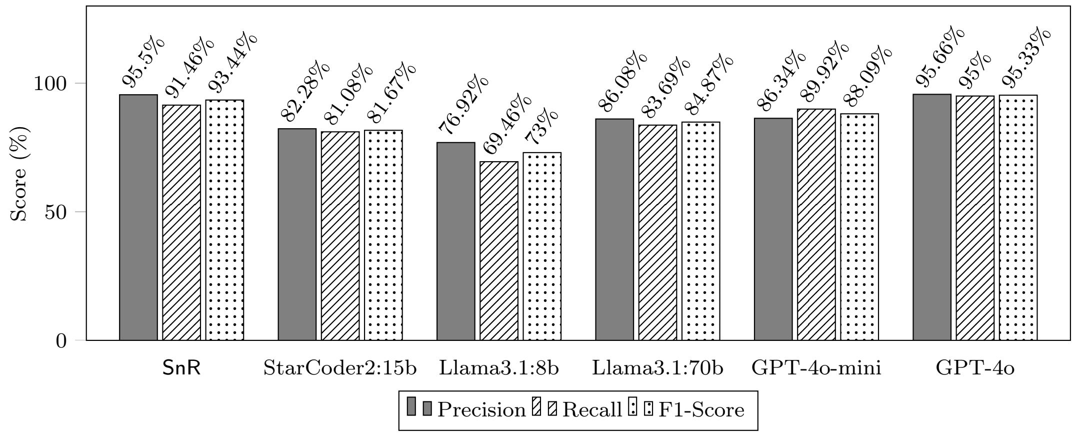
$$\text{Precision} = \frac{\text{Correctly Inferred FQNs}}{\text{All Inferred FQNs}} \quad \text{Recall} = \frac{\text{Correctly Inferred FQNs}}{\text{All Expected FQNs}} \quad \text{F1} = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$$

4.5.2 Results

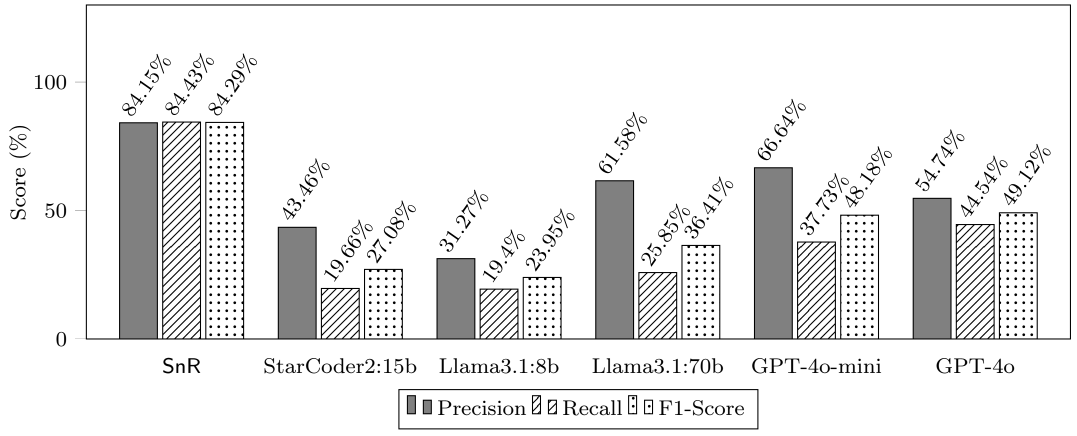
Despite achieving strong results in StatType-SO (Figure 4.6), all evaluated LLMs demonstrated lower performance on unseen code snippets in ThaliaType. StarCoder2:15b consistently ranked between Llama3.1:8b and Llama3.1:70b across both benchmark suites. Specifically, as shown in Figure 4.6a, StarCoder2:15b achieved an F1 score of 81.67% on StatType-SO, higher than Llama3.1:8b, but lower than Llama3.1:70b, GPT-4o, and GPT-4o-mini. However, its performance dropped sharply to 27.08% on ThaliaType, representing a 66.8% decrease. Despite this decline, it still outperformed Llama3.1:8b but remained below the other larger models. Notably, even the best performing model, GPT-4o, experienced a 48.5% decrease in F1 score when evaluated on ThaliaType. The consistent performance drop across all models, including StarCoder2:15b with confirmed data leakage, suggests that LLMs are highly likely affected by data leakage which potentially inflated LLMs' type inference performance on StatType-SO.

Unlike LLMs, SnR relies solely on the names of types, method names, and relationships between types in code snippets. SnR only experienced a 9.8% decrease in F1 score evaluating on ThaliaType, due to the difference in available information in the code snippets. The fact that LLMs outperformed SnR on StatType-SO while performing poorly on ThaliaType suggests that LLMs leverage additional information in StatType-SO to perform type inference. These performance discrepancies motivated further investigations into the specific factors contributing to LLMs' type inference capabilities. In particular, in RQ2 (§4.6), we investigate what types of information LLMs might leverage for type inference, and whether their performance depends on matching exact syntax of the code snippets seen during training.

Finding 1: LLM type inference performance declined dramatically when applied to generated, unseen code snippets, potentially as a consequence of data leakage. In these cases, LLMs performed substantially worse than the constraint-based method, highlighting a promising area for future improvement in LLM-based type inference.



(a) Type inference performance on StatType-SO.



(b) Type inference performance on ThaliaType.

Figure 4.6: Type inference performance (precision, recall, and F1-score) of SnR, StarCoder2:15b, Llama3.1:8b, Llama3.1:70b, GPT-4o-mini, and GPT-4o, on the StatType-SO and ThaliaType datasets. StarCoder2:15b performed in between Llama3.1:8b and Llama3.1:70b for both datasets. All LLMs experienced a large drop in type inference performance on ThaliaType compared to StatType-SO.

Table 4.3: Recall grouped by document frequency of FQNs on GitHub for types in StatType-SO and ThaliaType. The best result for each frequency group is colored using 🟦. The TP column lists the number of correctly inferred FQNs by each tool for each frequency group.

Document Frequency		[0,1e2)		[1e2,1e3)		[1e3,1e4)		[1e4,1e5)		>=1e5	
StatType-SO	Total FQNs	27		69		312		429		463	
		TP	Recall	TP	Recall	TP	Recall	TP	Recall	TP	Recall
	SnR	27	100.00%	60	86.96%	297	95.19%	372	86.71%	433	93.52%
	StarCoder2:15b	7	25.93%	36	52.17%	235	75.32%	372	86.71%	404	87.26%
	Llama3.1:8b	2	7.41%	28	40.58%	173	55.45%	322	75.06%	378	81.64%
	Llama3.1:70b	7	25.93%	31	44.93%	228	73.08%	383	89.28%	439	94.82%
	GPT-4o-mini	4	14.81%	43	62.32%	272	87.18%	399	93.01%	451	97.41%
	GPT-4o	6	22.22%	53	76.81%	296	94.87%	421	98.14%	459	99.14%
ThaliaType	Total FQNs	994		597		439		324		331	
		TP	Recall	TP	Recall	TP	Recall	TP	Recall	TP	Recall
	SnR	735	73.94%	533	89.28%	407	92.71%	300	92.59%	292	88.22%
	StarCoder2:15b	22	2.21%	58	9.72%	94	21.41%	120	37.04%	234	70.69%
	Llama3.1:8b	19	1.91%	42	7.04%	64	14.58%	138	42.59%	258	77.95%
	Llama3.1:70b	18	1.81%	80	13.40%	144	32.80%	182	56.17%	270	81.57%
	GPT-4o-mini	56	5.63%	161	26.97%	227	51.71%	269	83.02%	300	90.63%
	GPT-4o	103	10.36%	233	39.03%	256	58.31%	288	88.89%	316	95.47%

Impact of Document Frequency on LLM Type Inference

To better understand why there is a large disparity between StatType-SO and ThaliaType, we investigated how the document frequency of a type impacts the performance of LLMs. Document frequency measures the number of source files that reference a specific type. Since LLMs' outputs are biased by the training data [93, 94], it is plausible that their performance may vary depending on the prevalence of a type within an LLM's training data. Understanding this bias helps ensure that LLMs deliver similar performance across all potential types used in the real world. To quantify a type's document frequency, we analyzed source files from the Boa [95, 96] GitHub dataset. Specifically, the document frequency $F(T)$ is calculated as follows for a type T.

$$F(T) = \sum_{f \in \mathcal{F}} \begin{cases} 1 & \text{if } T \in f, \\ 0 & \text{otherwise.} \end{cases} \qquad (\mathcal{F} \text{ is the set of all source files in the GitHub dataset.})$$

Based on the document frequency, we categorized the FQNs in the StatType-SO and ThaliaType datasets into distinct frequency ranges. For each group, we calculated the recall to investigate how a type's real-world frequency impacts LLM performance. This categorization and analysis provide insights into the extent to which LLMs are biased toward more frequently encountered types.

Results. LLMs performed poorly on less frequently used FQNs and worse on the unseen dataset ThaliaType than on StatType-SO, even for frequently used FQNs. To analyze how performance varies across FQNs with differing levels of usage frequencies, we grouped FQNs into frequency ranges by orders of magnitude (0-100, 100-1,000, 1,000-10,000, 10,000-100,000, and over 100,000 GitHub files). For the least frequently used FQNs in ThaliaType, GPT-4o only achieved a recall of 22.22% on StatType-SO and 10.36% on ThaliaType. In contrast, `SnR` demonstrated consistent performance regardless of a type's popularity, achieving recalls of 100.00% and 73.94% on StatType-SO and ThaliaType respectively. On ThaliaType, LLMs only outperformed `SnR` on the most frequently used FQNs, specifically those appearing in over 100,000 GitHub source files. This disparity may arise from LLMs preferring more frequent FQN for a given simple name regardless of the context in the code snippet. For example, for a type with the simple name `View`, if `android.view.View` is the most frequent type in the >=1e5 category and `javax.swing.text.View` belongs to a less frequent category, then always recommending `android.view.View` will result in a high recall in the >=1e5 category but a lower recall in other categories.

Finding 2: LLM type inference performance diminishes when inferring less frequently used FQNs, highlighting a key limitation in applying LLMs to real-world applications. On unseen code snippets in ThaliaType, `SnR` outperformed all LLMs except for GPT-4o and GPT-4o-mini, but only for the most frequently used FQNs. This shortcoming may hinder LLMs' effectiveness for developers seeking assistance with these less common FQNs.

Interestingly, despite StarCoder2:15b being trained on code snippets from StatType-SO, it achieved a maximum recall of only 87.26%. As in the previous analysis, its performance generally fell between Llama3.1:8b and Llama3.1:70b on both StatType-SO and ThaliaType. Although training data may inflate performance, StarCoder2:15b does not reproduce code snippets verbatim, as models are designed to generalize beyond specific examples. This generalization, while desirable, makes it difficult to definitively detect instances of data leakage.

4.6 RQ2: To what extent do LLMs understand the execution semantics of code snippets?

The performance decline on unseen data observed in RQ1 highlights the need to investigate how LLMs handle type inference. To explore whether this decline stems from a lack of semantic understanding in LLMs, we conducted an experiment using semantic-preserving transformations that alter syntax without changing the semantics. Our hypothesis is that if LLMs possess a deep understanding of the code snippets' semantics, their performance should remain stable despite such transformations. Conversely, significant performance degradation would suggest that LLMs rely heavily on superficial syntactic patterns rather than a genuine semantic comprehension.

4.6.1 Method

The transformations, detailed in Algorithm 3, explore the extent to which syntactic changes affect LLMs' semantic understanding of code. Three specific transformations were employed to assess the impact of modifications to identifier names (line 35), code structures (line 35), and Java keywords in comments (line 35). These were selected to highlight potential limitations in LLMs' semantic understanding, though future research could explore additional transformations. Their impact was then evaluated using the methodology outlined in §4.5. The *Wilcoxon signed-rank test* [97] was used to determine whether the transformations caused significant changes in precision, recall, and F1 scores.

Background: Java Grammar

The Java grammar depicted in Figure 4.7 is used to illustrate the key transformations on the Java code snippet. This grammar is expanded from the one introduced in Figure 2.1 to include the rules for classes, modifiers, and statements. Each code snippet contains one or more classes along with a set of import statements. Classes may contain fields, methods, and blocks, with methods comprising blocks of statements and expressions. Highly repetitive Java statements and expressions were omitted as they follow the same transformation principles.

```
        Class  ::=  {Modifier} class SimpleName [extends Name] [implements {Name}]
                    ‘{’{ClassMember}‘}’
     Modifier  ::=  Annotation | public | protected | private | static | abstract | final
  ClassMember  ::=  Field | Method | Block
        Field  ::=  Type SimpleName [= Expr] ;
       Method  ::=  {Modifier} Type SimpleName ({Type SimpleName}) Block
   Annotation  ::=  @ Name
        Block  ::=  ‘{’{Statement}‘}’
    Statement  ::=  Expr ; | Expr = Expr ; | Block | return [Expr] ;
                     | Type SimpleName [= Expr] ;
                     | if ( Expr ) Statement else Statement
                     | while ( Expr ) Statement
                     | for ( Expr ; Expr ; Expr ) Statement
                     | for ( {Modifier} Type SimpleName [= Expr] ; Expr ; Expr ) Statement
         Expr  ::=  Name | Literal | this | super | Expr Op Expr
                     | ( Type ) Expr | Expr instanceof Name
                     | [Expr .] SimpleName | [Expr .] SimpleName ({Expr})
                     | new Name ({Expr }) | ! Expr | - Expr
                     | ( { [Type] SimpleName}) -> Block | SimpleName -> Block
                     | ( { [Type] SimpleName}) -> Expr | SimpleName -> Expr
      Literal  ::=  null | NumberLiteral | StringLiteral | BooleanLiteral
           Op  ::=  + | - | * | / | % | > | == | >= | !=
         Name  ::=  FQN | SimpleName
          FQN  ::=  Name . SimpleName
         Type  ::=  PrimitiveType | Name | var | void
PrimitiveType  ::=  int | boolean | float
```

Figure 4.7: Simplified Java grammar to illustrate our transformations. {*} denotes that the enclosed term occurs zero or more times. [*] denotes that the enclosed term occurs zero or one time.

Algorithm 3: Procedure for the three code transformations applied in RQ2.

```
def RenameIdentifier(p):
    Input  : p, representing the parsed code snippet.
    Output: Transformed code snippet with renamed identifiers.
    for variable_declaration in FindVariableDeclaration(p):          # Renaming all variables.
        random_name ← GetRandomName()
        ReplaceAllVariableName(p, random_name, variable_declaration.getName())
    skip_list ← { }
    for expression_method in FindExpressionMethodCalls(p):
        if expression_method.hasExpression():          # Skipping method calls with expression.
            skip_list.add(expression_method.getName())
    for method_declaration in FindMethodDeclaration(p):          # Renaming all method names.
        if method_declaration.getName() in skip_list or "@Override" in
          method_declaration.getAnnotations():
            continue          # Skipping methods that potentially override parent methods.
        random_name ← GetRandomName()
        ReplaceAllMethodName(p, random_name, method_declaration)
    RenameClassNames(p)                                              # Renaming class names.
    RenamePackageNames(p)                                            # Renaming package names.
    return p
def LowerCode(p):
    Input  : p, representing the parsed code snippet.
    Output: Transformed code snippet with lowered expressions.
    for expression in findExpressions(p):                            # Lowering expressions.
        if IsExprStatement(expression.parent()) or IsLoopCondition(expression) or
          IsLambdaExpression(expression):
            continue   # Skipping expression statements, loop conditions, or lambda expressions.
        random_name ← GetRandomName()
        InsertBefore("var { random_name } = { expression }", expression)
        Replace(p, random_name, expression)
    for field in findFields(p):                                      # Lowering field initializes.
        if field.hasInitialization():
            field_name ← field.getName()
            initializer ← field.getInitializer()
            field.removeInitializer()
            InsertBefore("{\n { field_name } = { initializer } ; \n }", field)
    return p
def AddKeyword(snippet):
    Input  : snippet, representing the original code snippet.
    Output: Code snippet with added keyword comments.
    new_lines ← []
    for line in snippet.split("\n"):                                 # Append a comment to every line.
        new_lines.add("{ line } //{ GetRandomKeyword() }")
    return new_lines
```

Identifier Renaming

This transformation investigates whether LLMs rely on specific identifier names for type inference. Although identifier names can provide useful signals, if a model relies on particular unique identifiers or specific identifier sequences seen during training, its generalizability may be compromised. In practice, however, identifiers in real-world code snippets are often lowercase variants of the type name or abbreviated forms, offering little additional semantic information.

The `RenameIdentifier` function in Algorithm 3 details the process, which systematically renames identifiers such as variables, methods, classes, and packages in the code snippet. First, all variable declarations are identified using the `FindVariableDeclaration` function (line 2), which extracts statements matching the pattern *Type* *SimpleName* [`=` *Expr*] `;`. The algorithm then traverses each declaration, generating a unique, random three-word name for each variable using the `GetRandomName` function (line 3), from a predefined word list from prior work [90] to ensure uniqueness. The algorithm replaces every occurrence of the variable name in the code (extracted using the `getName` function in line 4).

Next, the algorithm processes method declarations. To avoid renaming references that could alter semantics, all the method call expressions (*Expr*s with the form [*Expr* `.`] *SimpleName* `(`{*Expr*}`)`) that have the first optional part ([*Expr* `.`]) is potentially external, thus added to a `skip_list` (line 7-8) to be filtered out (line 10). Additionally, overridden methods annotated with `@Override` are skipped, as their names must match those in the super class (line 10). Lastly, the algorithm renames classes and packages (lines 14-15), following the same procedure used for variables. The details are omitted here for brevity.

Figure 4.8 provides an example of this transformation, showing how variables, methods, classes, and packages are renamed. Method `m` which likely overrides the method `m` on type `a` is preserved out of an abundance of caution to preserve semantic consistency. Furthermore, it should be noted that different types of names are renamed separately. For example, the method name `n` is renamed to a different name than the variable name `n`.

Code Lowering

The code lowering transformation examines the impact of structural changes on LLMs' type inference. A key aspect of type inference involves understanding the types present in a program and how they relate to one another. Code lowering does not alter these type relationships but instead disrupts the exact sequence of tokens presented to the model. If a model relies heavily on specific token sequences seen during training, its generalizability

```
package p;

class C extends A {
  A a;
  void m() {
    a.m();
  }
  int n() {
    int n = 1;
    return 1;
  }
}
```

(a) Before renaming.

Before	After
p	GrouseScabsShelley
C	TashaMonroviaTimbers
a	RowsGarlickyThump
n()	ListerineStupefiesFetlock()
n	CrackerSherbertsPlod

(b) Renamed identifiers.

```
package GrouseScabsShelley;

class TashaMonroviaTimbers extends A {
  A RowsGarlickyThump;
  void m() {
    RowsGarlickyThump.m();
  }
  int ListerineStupefiesFetlock() {
    int CrackerSherbertsPlod = 1;
    return 1;
  }
}
```

(c) After applying identifier renaming.

Figure 4.8: Example identifier renaming on a simplified code snippet.

may be impaired, leading to failure in inferring types when the same program is presented in a different structure.

The algorithm first extracts and traverses all the expressions (line 18) in the code snippet. For each expression, if its parent is not an expression statement (*Statements* with the form *Expr* ;) and it is not a loop condition (lines 19-20), the algorithm creates a new variable with a random name, assigns the expression to the variable, and inserts this assignment statement before the expression (lines 21-22). Next, the original expression is replaced with the newly created variable assigned with the value of the original expression (line 23). Such transformations modify the code structure while preserving the semantics of the code snippet. The expressions in expression statements must be skipped because transforming them may result in invalid code. For example, an expression in an expression statement can be a method call with no return value (*i.e.*, with a `void` return type). In such a case, the created assignment statement would be invalid. Conditions in loops cannot be transformed either since such transformations could change the semantics of the code.

```java
class C {
  C c = null;
  void m() {
    int a = 0;
    if (a == 0) {
      return;
    }
  }
}
```

(a) Before code lowering.

```java
class C {
  {
    c = null;
  }
  C c;
  void m() {
    int a = 0;
    var GrouseScabsShelley = a == 0;
    if (GrouseScabsShelley) {
      return;
    }
  }
}
```

(b) After applying code lowering.

Figure 4.9: Example code lowering on a simplified code snippet.

In addition to expressions, code lowering also applies to fields with an initializer. For each field in the code snippet (line 24), if it has an initializer (line 25), the algorithm splits it into a declaration and a block initializing the field (lines 26-29).

Figure 4.9 demonstrates this transformation. In this example, the field `C c = null` is split into a declaration and an initializer block with `c = null`. Additionally, the expression in the `if` statement is replaced with a newly created variable, and an assignment statement is inserted before the expression's usage to assign its original value to the variable.

Adding Keyword Comments

The keyword comment transformation evaluates whether comments containing Java keywords distract LLMs. If a model relies on specific sequences of keywords seen during training, rather than on the actual content of the code, then this transformation may lead to reduced performance and an inability to generalize to unseen code snippets. To perform this transformation, the `AddKeyword` function in Algorithm 3 appends a single Java keyword [98] as a line comment at the end of each line in the code snippet. An example of the transformed snippet is presented in Figure 4.10.

Putting Everything Together

Figure 4.11 illustrates the complete set of transformations used in RQ2. Each of the first three transformations was applied independently to measure its individual impact. The final transformation combines all three sequentially to more strongly perturb the input,

```
class C {
  void m() {
    return;
  }
}
```

(a) Before adding keyword comments.

```
class C { // continue
  void m() { // void
    return; // int
  } // opens
} // case
```

(b) After adding keyword comments.

Figure 4.10: Example of adding keyword comments on a simplified code snippet.

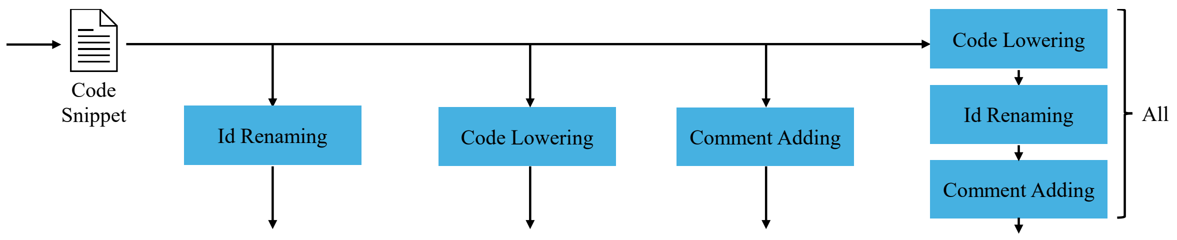


Figure 4.11: Overview of the transformation workflow for RQ2. Three code transformations were designed and applied individually to assess their isolated impact. A fourth transformation combines all three sequentially to evaluate potential non-linear interactions.

thereby making it more difficult for models to directly recall StatType-SO code snippets, if they were seen during training.

4.6.2 Results

The effects of applying transformations to the StatType-SO and ThaliaType datasets are shown in Tables 4.4a and 4.4b, respectively. These transformations significantly affected the performance of all evaluated LLMs, causing significant perturbation in precision, recall, and F1-scores. However, the results are not straightforward and merit detailed examination.

Focusing on each transformation in isolation (identifier renaming, code lowering, comment adding), the results show that LLMs generally exhibit resilience, meaning that simple transformations often do not significantly change a model's performance. This was particularly true for more advanced models such as GPT-4o but also held for smaller models like StarCoder2:15b. Notably, StarCoder2:15b, which is known to have been trained on StatType-SO code snippets, exhibited significant performance drops only under identifier renaming (in recall) and comment adding (in recall and F1-score) when evaluated on StatType-SO. The isolated transformation results on StatType-SO and ThaliaType indicate a degree of generalizability in LLMs' understanding of code.

Table 4.4: The precision, recall, and F1-scores of tools after transformations identifier renaming (Id Renaming), code lowering, and adding keyword comments (comment adding) were applied. Note that ⬜ represents $p < 0.05$, ▨ represents $p < 0.01$, and 🟦 represents $p < 0.001$ when performance decreases, and 🟫 represents $p < 0.05$, and 🟧 represents $p < 0.001$ when performance improves.

(a) The precision, recall, and F1-scores of tools on StatType-SO and transformed code snippets.

	SnR			StarCoder2:15b			Llama3.1:8b			Llama3.1:70b			GPT-4o-mini			GPT-4o		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
StatType-SO	95.50%	91.46%	93.44%	82.28%	81.08%	81.67%	76.92%	69.46%	73.00%	86.08%	83.69%	84.87%	86.34%	89.92%	88.09%	95.66%	95.00%	95.33%
Id Renaming	95.50%	91.46%	93.44%	82.09%	71.23%	76.28%	65.28%	56.69%	60.68%	85.34%	76.15%	80.49%	89.63%	88.46%	89.04%	97.03%	95.62%	96.32%
Code Lowering	95.43%	91.62%	93.49%	83.72%	79.92%	81.78%	69.43%	65.15%	67.22%	83.81%	80.46%	82.10%	86.14%	88.92%	87.51%	94.82%	95.69%	95.25%
Comment Adding	95.50%	91.46%	93.44%	83.98%	75.00%	79.24%	78.65%	67.46%	72.63%	85.39%	80.92%	83.10%	87.90%	89.38%	88.63%	96.06%	95.77%	95.92%
All	95.43%	91.62%	93.49%	79.34%	59.38%	67.93%	55.83%	47.15%	51.13%	77.93%	70.08%	73.80%	84.60%	82.85%	83.72%	93.29%	94.15%	93.72%

(b) The precision, recall, and F1-scores of tools on ThaliaType and transformed code snippets.

	SnR			StarCoder2:15b			Llama3.1:8b			Llama3.1:70b			GPT-4o-mini			GPT-4o		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
ThaliaType	84.15%	84.43%	84.29%	43.46%	19.66%	27.08%	31.27%	19.40%	23.95%	61.58%	25.85%	36.41%	66.64%	37.73%	48.18%	54.74%	44.54%	49.12%
Id Renaming	84.15%	84.43%	84.29%	46.24%	22.42%	30.20%	31.36%	19.11%	23.74%	57.08%	24.47%	34.25%	52.47%	38.44%	44.37%	43.88%	45.77%	44.80%
Code Lowering	84.14%	84.39%	84.27%	48.09%	23.87%	31.91%	27.06%	18.21%	21.77%	60.16%	25.25%	35.57%	58.04%	36.01%	44.45%	43.45%	44.43%	43.93%
Comment Adding	84.15%	84.43%	84.29%	42.96%	19.66%	26.98%	25.00%	19.22%	21.73%	59.56%	25.07%	35.28%	64.27%	36.57%	46.62%	32.26%	44.80%	37.51%
All	84.14%	84.39%	84.27%	40.81%	20.74%	27.51%	27.57%	17.77%	21.61%	45.92%	23.87%	31.41%	47.50%	36.87%	41.52%	56.30%	45.29%	50.20%

Finding 3: LLMs, especially more advanced models, demonstrate generalizability to simple transformations that preserve execution semantics, maintaining overall performance despite syntactic changes such as identifier renaming, code lowering, and comment adding. This shows that the models tested do not rely on any one syntactic element for type inference.

However, when all three transformations were applied simultaneously, the changes in performance for LLMs were amplified and clearly, significantly, negatively impacted all LLMs' precision, recall, and F1-scores on StatType-SO. For example, F1-scores dropped by 1.7% on GPT-4o, 16.8% on StarCoder2:15b, and 30.0% on Llama3.1:8b. Similar trends were observed for precision and recall. In contrast, applying the same three transformations on ThaliaType code snippets produced less consistent and generally smaller effects. In some cases, performance even improved. For instance, StarCoder2:15b achieved a higher F1-score despite a lowered precision, and GPT-4o demonstrated higher precision, with no significant changes in other metrics on ThaliaType code snippets.

Focusing on GPT-4o on StatType-SO, despite identifier renaming and comment adding causing insignificant performance increases, and code lowering causing a small but significant decrease in precision without impacting F1, when all combined, significantly decreased performance in precision, recall, and F1-scores. These effects were not observed for any model using ThaliaType code snippets.

The discrepancies in performance indicate that the results in StatType-SO may be influenced by data leakage and have limited generalizability compared to the results in ThaliaType. A potential explanation is that transformations reduced the ability of LLMs to recognize code patterns from StatType-SO snippets from training, leading to degraded performance. In contrast, ThaliaType consists of newly generated code snippets that were not in the training data. As a result, models must rely more on reasoning over execution semantics rather than recall from training. The observation that performance on ThaliaType does not consistently decline, and sometimes even improves, suggests that current LLMs can withstand simple, execution-semantic preserving transformations, and demonstrate greater generalizability than LLMs' performance on StatType-SO.

In contrast, SnR, which analyzes the semantic meaning behind the code snippets, was unaffected by these transformations, experiencing only minor, statistically insignificant variations under the code lowering transformation. These variations occurred when multiple types satisfied all constraints, with the order of type variables serving as a tie-breaker.

Finding 4: In contrast to SnR, which is designed to extract semantic meaning from code snippets and remains unaffected by semantic-preserving transformations, LLMs' type inference generalizability is negatively impacted by combined transformations on StatType-SO, a pattern not observed in ThaliaType. This contrast suggests that while LLMs are capable of generalizing in ThaliaType and its transformed variants, their generalization ability diminishes in StatType-SO, potentially due in part to data leakage.

4.7 Threats to Validity

There are four main threats to validity.

First, the **representativeness of ThaliaType** may not fully capture the diversity and complexity of real-world Java code snippets. However, this limitation is mitigated by the primary objective of our evaluation, which is to assess the type inference capabilities of LLMs. Specifically, ThaliaType is designed to evaluate whether LLMs can infer correct type information by reasoning about the execution semantics of the code snippets, rather than merely recalling examples from training. We acknowledge that LLMs may leverage additional semantic cues present in StatType-SO but not found in ThaliaType for type inference. To account for these potential differences, RQ2 applies transformations to both datasets. Regardless, there were still significant performance decreases on StatType-SO code snippets. Furthermore, the strong performance of SnR on both StatType-SO and ThaliaType indicates that, (1) ThaliaType shares key features that are necessary for type inference with StatType-SO, (2) it presents a meaningful challenge for type inference techniques, and (3) it exposes areas where LLMs require further improvement.

Second, variations in the **document frequency of FQNs** between ThaliaType and StatType-SO datasets could influence LLM type inference performance, since LLMs' outputs are biased by the training data [93, 94]. LLM's performance may vary depending on the prevalence of a type within LLM's training data. To address this, we analyzed the performance of LLMs across different document frequency levels. Despite these efforts, LLMs consistently exhibited lower performance on unseen code snippets from ThaliaType compared to StatType-SO.

Third, our findings may **not generalize to other LLMs**. We mitigated this by following best practices [50] and evaluated a diverse set of models, including both open-weight (StarCoder2:15b, Llama3.1:8b, Llama3.1:70b) and state-of-the-art closed models (GPT-4o, GPT-4o-mini) of varying sizes, providing a representative sample of current LLM capabilities.

Fourth, **prompt engineering** may enable LLM to achieve better performance for type inference on code snippets. To mitigate this threat, we followed best practices to design a simple but effective prompt [88], which achieved high performance on the StatType-SO dataset. Regardless, data leakage remains a fundamental issue that can influence LLM evaluations regardless of prompt quality. Our evaluation highlights how data leakage may have impacted prior assessments of LLM type inference performance.

4.8 Related Work

In this section, we briefly discuss different techniques for type inference on code snippets (particularly machine-learning-based approaches) and LLM data leakage in other fields, including automated program repair, code generation, refactoring, and more.

4.8.1 Type Inference for Code Snippets

Existing type inference approaches for code snippets mainly fall into two categories, constraint-based approaches and machine-learning-based approaches.

Constraint-Based Approaches. The general idea of constraint-based approaches is to first analyze the code snippet and derive a collection of constraints on the types that need to be inferred. With the derived constraints, a set of APIs that satisfies all the constraints can be obtained by performing a constraint-solving algorithm. Baker is the first work that applied constraint-based type inference for code snippets [1]. SnR, proposed in §3, outperformed Baker by improving the handling of parameterized types and leveraged constraint solving for fast and efficient type inference. Thus, SnR is used as a baseline to compare LLMs' performance.

Machine-Learning-Based Approaches. Machine-learning-based approaches typically leverage a model trained with a large set of programs from open-sourced projects. Phan et al. [2] proposed StatType, which learns the FQNs that often co-occur from a large corpus. With such knowledge, StatType can derive the FQN for an API based on the neighboring API names. A subsequent work conducted by Saifullah et al. [3] made improvements by leveraging both local and global contexts. Huang et al. [5] leveraged pre-trained CodeBert [99], a transformer-based masked language model to predict FQNs in the code snippet. Compared to LLMs evaluated in this study, CodeBert is much smaller with 125 million parameters, compared to even Llama3.1:8b's 8 billion parameters. Kabir et al. [7] extended the single-prompt approach by incorporating a secondary prompt to address compilation errors in the code snippet generated by the first prompt. However, since our work focuses on mitigating data leakage, which fundamentally undermines inference reliability regardless of the prompting strategy used, we used a single prompt. Chen et al. [8] proposed a hybrid approach that iteratively combines constraint-based techniques, such as SnR, with machine learning methods to refine the results further. Nevertheless, because machine learning models are used to refine outputs produced by SnR, this hybrid approach remains susceptible to data leakage.

4.8.2 Type Inference for Dynamically Typed Languages

Some dynamically typed languages also support compile-time type checking by adding type annotations. For example, in Python 3.5, a library named typing is introduced to support type hints; TypeScript enhances JavaScript with a type system by allowing variables to have a type annotation. Type inference for dynamically typed languages presents additional challenges as dynamically typed languages often contain variables with insufficient static type constraints, which static type inference cannot handle soundly. Many existing studies resort to deep learning-based approaches to overcome this challenge. For example, Hellendoorn et al. [72] propose DeepTyper, a deep learning model trained to provide type suggestions based on contexts and relations. Malik et al. [73] propose NL2Type, a machine-learning approach that utilizes natural language information to predict type annotations. Pradel et al. [100] propose TypeWriter, which employs a deep learning-based approach to predict types and then utilizes a search-based approach to validate the predicted types. Peng et al. [101] propose HiTyper, a deep learning-based approach that combines static type inference. It conducts static inference and DL-based prediction iteratively to construct a type dependency graph, which records type dependency information among variables. This thesis focuses on the performance of LLMs in type inference for Java code snippets, leaving the investigation of type inference for dynamically typed languages to future work.

4.8.3 LLM Data Leakage

Data leakage is not limited to type inference but also affects other software engineering tasks. Currently, LLMs have been integrated into a wide variety of software engineering tools such as automated code repair [29, 30, 31, 32, 33, 34, 35, 36], code generation [37, 38, 39, 40], code refactoring [41, 42], and code completion [43, 44, 45], which can all be potentially affected by data leakage either currently or in the future.

Xia et al. [31] discovered that 15% of the bug-fixing patches for automated program repair generated by earlier LLMs (*i.e.*, CodeT5, GPT-Neo, GPT-J, and GPT-NeoX) were already present in the training data. Sainz et al. [46, 102] examined several academic datasets and reported that most of them were either used as training data for ChatGPT or likely exposed to it. A subsequent work done by Golchin and Surdeanu [103] proposes a more advanced detection technique and detects the presence of test data of several datasets in the training data of LLMs. Recently, Kong et al. [51] detected memorization in LLMs using low-probability events such as exact code repair matches. However, for type inference on code snippets, there is generally one ground truth of types that are expected. Thus, exact matches do not precisely indicate data leakage. As LLM training datasets grow larger and increasingly opaque, detecting instances of data leakage during evaluation becomes progressively more challenging.

The data leakage problem remains an open challenge for evaluating LLMs. Some studies have been conducted in other fields [104, 105, 106, 107, 108, 109] regarding data leakage and LLM evaluation. Mirzadeh et al. [106] showed that LLMs have significant limitations when conducting genuine mathematical reasoning, which suggests that LLMs are conducting sophisticated pattern matching rather than true logical reasoning. A recent study by

Cao et al. [107] demonstrated that the performance of LLMs is far behind undergraduate students on the proposed project-level Java benchmark that exercises object-oriented programming features.

Our evaluation on type inference rather than math questions or code generation also showed a significant drop in performance with generated code snippets which suggests that understanding the type system in code is challenging for LLMs. So far, there is no conclusive evidence showing that LLMs are conducting true type inference on code snippets. Small changes to the inputs can drastically alter model outputs [110, 106, 111]. While some guidelines have been raised for software engineering research using LLMs [86, 50], protecting against data leakage remains an open challenge. We hope our work will shed further light on the data leakage issue in software engineering.

4.9 Chapter Conclusion

This chapter conducted a comprehensive assessment of LLMs' type inference capabilities on Java code snippets. Our comprehensive evaluation mitigated potential data leakage issues and identified possible limitations of LLMs. First, we create a new dataset named ThaliaType. By evaluating the performance of LLMs on ThaliaType and StatType-SO, we found that all LLMs suffer similar degradations in performance on unseen code snippets, consistent with the LLM with confirmed data leakage, resulting in up to 59% decrease in precision and up to 72% decrease in recall. In addition, LLM type inference performance diminishes when inferring less commonly used FQNs, potentially introducing bias and presenting opportunities for future research to enhance performance. Second, we designed and applied three semantic-preserving code transformations to code snippets in ThaliaType and StatType-SO to investigate LLMs' understanding of the execution semantics. Through evaluating LLMs with these transformed code snippets, we find that while LLMs exhibit consistent performance under simple transformations, the consistency is not maintained in combined transformations on StatType-SO. The performance degradation observed using combined transformations on StatType-SO suggests that the LLMs' performance on that dataset may not be representative of their actual performance, potentially due to data leakage. Our findings suggest that future evaluations of LLMs should also incorporate unseen datasets, such as ThaliaType and transformed variants, rather than relying solely on StatType-SO, to better assess the generalization capabilities of LLMs in type inference tasks. All of our code and our dataset are available at `https://github.com/uw-pluverse/thalia-type`.

Chapter 5

Scitix: Scalable Constraint-Based Type Inference for Code Snippets with Unknown Types

This chapter builds on the work presented in §3 and addresses the scalability challenge that arises in code snippets with unknown types.

5.1 Introduction

Recall back from §3, SnR uses constraint solving to find in totality the set of solutions that satisfy all the constraints extracted from the code snippet, if such a solution exists. That is why, the more libraries that are included in its knowledge base, the more likely that SnR will be able to infer arbitrary code snippets regardless of the types and libraries that snippet might refer to. Practical, real-world deployment of a constraint-based type inference technique needs to include thousands of jars in the knowledge base to effectively support arbitrary code snippets in the wild [78].

While SnR is efficient when the size of the knowledge base is small (*e.g.*, with 49 jars from StatType-SO), its efficiency drastically decreases when the knowledge base expands. This limitation stems from SnR improperly handling the *unknown types*, *i.e.*, types in the code snippet that no type in the knowledge base can fit in. When there are unknown types, SnR cannot find a set of types that satisfies all the extracted constraints. In such a scenario, SnR instead attempts to enumerate and validate large possible subsets of constraints

```
public void to_reminder(View view) {
    Intent intent = new Intent(this, Notification_morning.class);
    AlarmManager manager = (AlarmManager) getSystemService(Activity.ALARM_SERVICE);
    PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);

    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.HOUR_OF_DAY, timepicker.getCurrentHour());
    cal.set(Calendar.MINUTE, timepicker.getCurrentMinute());
    cal.set(Calendar.SECOND, 0);
    cal.set(Calendar.MILLISECOND, 0);
    manager.setRepeating(AlarmManager.RTC_WAKEUP, cal.getTimeInMillis(),
                         24*60*60*1000, pendingIntent);
}
```

Figure 5.1: Formatted Stack Overflow code snippet #14241439 showing an event handler method called by Android with the relevant but unused `View` object.

using a series of heuristics. As the knowledge base expands, the time required to check whether a subset of constraints is satisfiable increases, which in turn lengthens the overall inference time. For example, consider the code snippet in Figure 5.1, where types `this`, `Notification_morning`, and `timepicker` are immediately identifiable as unknown. In this case, SnR takes 49 minutes to enumerate possible solutions using a large knowledge base of over 3,000 jars, which is far too impractical for real-world use.

Our Approach. To make constraint-based type inference *scalable* and *precise* in practice in the presence of large knowledge bases and unknown types, we propose Scitix, a new constraint-based type inference technique, which is based on the following two key, novel components:

1. Inspired by ideas from gradual typing [112], we propose a special `Any` type to represent the unknown types within code snippets as a means to escape the strict constraint-based type inference. The `Any` type is a special type that can be implicitly converted to any other type, and all other types can also be implicitly converted to `Any`. Our technique marks types explicitly referenced in a code snippet, but *unknown* to the knowledge base, as `Any`.

2. We devise an iterative approach to add constraints to improve the initial solution with `Any` types to increase the precision of Scitix while improving scalability by skipping constraints with unknown types that cannot be satisfied.

When given a code snippet, Scitix automatically recommends dependencies and FQNs for the types used. Despite being more flexible, Scitix remains highly *precise* and *explainable* in its inference process, especially when compared to prior machine learning-based approaches [2, 3, 5, 6].

We extensively evaluated Scitix against state-of-the-art tools using manually repaired Stack Overflow code snippets (StatType-SO), a ubiquitous benchmark suite used to evaluate prior work [4, 2, 3, 5, 6, 7, 8]. Our evaluation compares Scitix against `SnR`, a state-of-the-art constraint-based type inference tool, as well as state-of-the-art LLMs that have shown promise in type inference from prior work [7, 8, 5]. However, due to concerns with *data leakage*, where LLMs likely have been exposed to StatType-SO code snippets during training, we leveraged ThaliaType proposed in §4 for evaluation. Across both StatType-SO and ThaliaType, Scitix outperformed all state-of-the-art tools, including LLMs. Using the largest knowledge base, which comprised over 3000 jars, led `SnR` to timeout on most code snippets, yielding an F1-score that is close to zero. In contrast, Scitix exhibited consistently high performance, achieving an F1-score of 96.6% on StatType-SO and 88.7% on ThaliaType. Even on the smallest knowledge base, compared to `SnR`, Scitix reduced the number of errors in the recall by 79% and 37% for StatType-SO and ThaliaType, respectively. Against LLMs, Scitix outperformed all models on StatType-SO using the largest knowledge base, despite the potential for data leakage, and markedly outperformed them on ThaliaType. In particular, Scitix reduced the number of errors in the recall by 20% on StatType-SO and 78% on ThaliaType compared to GPT-4o.

Contribution. We make the following major contributions.

- We propose Scitix, a novel, scalable approach that efficiently and effectively handles unknown types in constraint-based type inference on code snippets using the `Any` type and an iterative approach.
- We evaluate Scitix against real-world code snippets from StatType-SO and the data leakage resistant ThaliaType dataset. Compared to the state-of-the-art `SnR`, Scitix reduces the errors in recall by 79% on StatType-SO and 37% on ThaliaType.
- We demonstrate that Scitix outperforms state-of-the-art LLMs, reducing the error rate by 20% on StatType-SO and 78% on ThaliaType.
- For reproducibility and replicability, we have provided a replication package at `https://figshare.com/s/f03c5103e2ab02125b83`.

5.2 Motivating Example

In this section, we illustrate how SnR would infer the types in the code snippet, the limitations of SnR (§5.2.1), and a sketch of the solution (§5.2.2). We use Figure 5.1 as our motivating example. To reuse this code snippet, developers must:

(a) Identify the FQNs of the six simple names referring to types from popular libraries.
(b) Add the declaration and initialization for the variable `timepicker`.
(c) Recreate the `Notification_morning` class, written by the code snippet author.

This thesis and prior work [1, 2, 3, 4, 89, 6] focus on step `(a)` which can assist the developers in reusing this code snippet. However, all prior work has overlooked the additional challenges posed by unknown types, such as `timepicker` and `Notification_morning`, which complicate step `(a)`.

To recap a little bit, SnR infers the types in Figure 5.1 following a three-step process. First, it extracts the constraints from the code snippet through static analysis. Second, it queries a knowledge base using these constraints. Third, it solves the constraints with the information found in the knowledge base using constraint solving. To illustrate this process, we examine the following two statements from Figure 5.1.

```
Calendar cal = Calendar.getInstance();
cal.getTimeInMillis();
```

Extract Constraints. To create constraints, types used in the code snippet are represented with *type variables* (TVs), which are then solved to find the types that satisfy the constraints for each TV. The previous expressions would then be labeled as follows.

```
τ1 cal = τ1.getInstance();
τ1.getTimeInMillis();
```

One can consider constraint solving as the process of finding types from the knowledge base to replace the type variables that satisfy all the constraints.

From the two statements for `Calendar`, the following constraints state that τ_1 has a simple name `Calendar` and methods `getInstance`, `getTimeInMillis`. The wildcard symbol `"_"` indicates that the FQN of `Calendar` is unknown. The `method_id` variables are used to match identifier numbers that uniquely distinguish methods in the knowledge base.

Table 5.1: Query result for FQNs using the simple names in Figure 5.1 using the largest knowledge base with over 3000 most popular jars. The correct FQNs are shown in bold. ∅ indicates there is no type in the knowledge base with the given simple name. Portions of the longer FQNs have been replaced by . . . to ease presentation.

Simple Name	FQN
`AlarmManager`	`android.app.AlarmManager`
`PendingIntent`	`android.app.PendingIntent`
`View`	`android.view.View`
	`javax.swing.text.View`
	`org.hsqldb.View`
	`org.springframework...View`
	... 51 more rows omitted
`Intent`	`com.lambdaworks.redis...Intent`
	`android.content.Intent`
`Activity`	`com.ibm.sbt.services...Activity`
	`eu.agrosense.api.task.Activity`
	`android.app.Activity`
`Calendar`	`java.util.Calendar`
	`org.quartz.core.Calendar`
	`org.elasticsearch...Calendar`
	... 3 more rows omitted
`timepicker`	∅
`Notification_morning`	∅

- class(τ_1, _, "Calendar")
- method(method_id$_1$, τ_1, τ_2, "getInstance")
- method(method_id$_2$, τ_1, τ_3, "getTimeInMillis")

Search for Related Types in the Knowledge Base. To find the FQNs for the TVs in the code snippet, SnR would query the knowledge base with the information from the extracted constraints to find related types that the TVs might represent. For τ_1, SnR would query for a type with a simple name `Calendar`. Table 5.1 shows results from querying the knowledge base using the simple names in Figure 5.1 (note that names in Java are case-sensitive). These queries generally result in one of the following scenarios.

① Returning no FQNs. In this case, it can be reasonable to conclude that the searched for type is unknown (*e.g.* `timepicker`, `Notification_morning`).

② Returning one or more FQNs. In this case, type inference is needed to determine which, if any of the returned FQN is correct.

For `Calendar`, the query resulted in scenario ② with a number of different `Calendar` types from multiple libraries. SnR leverages constraint solving to narrow down the set of FQNs found in the knowledge base.

Solve Constraints. Below is a simplified list of facts used for solving constraints. To improve clarity, other relevant classes and their respective methods have been omitted. These facts specify that the Java Standard Library (`jdk`) contains a class with the FQN `java.util.Calendar`, which has the simple name `Calendar`. This class has the methods `getInstance` and `getTimeInMillis`, which return `Calendar` and `long` types, respectively. Additionally, the `sundial` library contains a class with the FQN `org.quartz.core.Calendar`, which has the simple name `Calendar`, but without relevant methods.

```
– class("jdk:java.util.Calendar", "java.util.Calendar", "Calendar")
– method(0, "jdk:java.util.Calendar", "jdk:java.util.Calendar", "getInstance")
– method(0, "jdk:java.util.Calendar", "jdk:long", "getTimeInMillis")
– class("sundial:org.quartz.core.Calendar", "org.quartz.core.Calendar", "Calendar")
```

These facts and the earlier constraints are then processed by a constraint solver to compute a mapping from each TV to a type in the knowledge base. Since the `Calendar` type from `sundial` lacks the `getInstance` and `getTimeInMillis` methods, the constraints precisely determine that the only compatible type for τ_1 is $\tau_1 \rightarrow$ `"jdk:java.util.Calendar"`. Using this mapping, SnR can correctly add `import java.util.Calendar`.

However, the presence of unknown types makes the entire set of constraints unsatisfiable. While the type under scenario ① can be reasonably considered unknown, finding exactly which type is unknown under scenario ② is impossible without the ground truth. The way SnR addresses the issue is to guess which type is unknown based on a series of heuristics. However, when the guess is wrong, the precision and recall of the results decrease. Moreover, when the knowledge base size increases, more simple names match to one or multiple FQNs (*i.e.*, fall in scenario ②), and it becomes more difficult for SnR to guess the unknown type correctly, thus further losing precision and recall.

5.2.1 Limitations of SnR

SnR models type inference as a Constraint Satisfaction Problem (CSP) [113]. Given a set of FQNs and a set of constraints, a CSP solver finds a set of FQNs that satisfy all the constraints in a code snippet, or determines that the constraints are unsatisfiable when no solution exists. However, CSP is unable to handle unknown types, which leads to reduced precision in inference. We illustrate this issue using line 2 in Figure 5.1, reproduced below.

```
new Intent(this, Notification_morning.class);
```

To try and solve for the types in the code snippet using constraint-based type inference, again we annotate the code snippet with the TVs representing the used types.

```
new τ₁(τ₂, τ₃);
```

Then, SnR extracts a set of constraints where τ_1, τ_2, and τ_3 correspond to the simple names `Intent`, `Main`, and `Class`, respectively. This assumes the code snippet is wrapped inside a class named `Main` to ensure it is parsable. In addition, a method constraint shows that the constructor of `Intent` (represented as `<init>`) takes two parameters, τ_5, and τ_6, which must be supertypes of τ_2 and τ_3.

```
– class(τ₁, _, "Intent")                    – method_param(method_id₁, 0, τ₅)
– class(τ₂, _, "Main")                      – method_param(method_id₁, 1, τ₆)
– class(τ₃, "java.lang.Class", "Class")     – supertype(τ₅, τ₂)
– class(τ₄, "void", "void")                 – supertype(τ₆, τ₃)
– method(method_id₁, τ₁, τ₄, "<init>")
```

To solve these constraints, facts are again retrieved from the knowledge base. However, even with an oracle providing the correct type for `Intent`, the set of constraints above remains unsatisfiable. Here are the relevant facts for `Intent`.

```
– class("android:android.content.Intent",  – method_param(0, 0,
  "android.content.Intent", "Intent")        "android:android.content.Context")
– class("jdk:java.lang.Class",              – method_param(0, 1,
  "java.lang.Class", "Class")                "jdk:java.lang.Class")
– class("jdk:void", "void", "void")         – supertype("jdk:java.lang.Class",
– method(0, "android:android.content.Intent""jdk:java.lang.Class")
  "jdk:void", "<init>")
```

From the facts, we cannot find valid types for all TVs.

τ_1	$\rightarrow$	"android:android.content.Intent"	τ_4	$\rightarrow$	"jdk:void"
τ_2	$\rightarrow$	??	τ_5	$\rightarrow$	"android:android.content.Context"
τ_3	$\rightarrow$	"jdk:java.lang.Class"	τ_6	$\rightarrow$	"jdk:java.lang.Class"

Specifically, no valid type can be found for τ_2 because there is no correct `Main` type in the knowledge base. Thus, a CSP solver would return *unsatisfiable* for the set of constraints from Figure 5.1. So then, how can we determine that the type `Intent` in the code snippet comes from the `Android` library? After all, the simple name `Intent` can also be from `Redis` as seen from Table 5.1 as `com.lambdaworks.redis...Intent`. Types from both `Android` and `Redis` libraries do not satisfy the constraints, and thus, it is infeasible for CSP solvers to determine which one is more suitable. Besides, it is also possible that `Intent` is a user-defined type that does not exist in the knowledge base.

5.2.2 A Glance at Scitix

Although the constraints extracted from Figure 5.1 are unsatisfiable as a whole, we observe that the majority of individual constraints still provide value when inferring the types in the code snippet. Since we, acting as human experts, know that the correct solution involves ignoring `Main`, we strike through and disregard its related constraints. This allows us to focus on constraints for the class, method, and remaining method parameter of `Intent`.

- `class(`τ_1`, _, "Intent")`
- ~~`class(`τ_2`, _, "Main")`~~
- `class(`τ_3`, "java.lang.Class", "Class")`
- `class(`τ_4`, "void", "void")`
- `method(method_id`$_1$`, `τ_1`, `τ_4`, "<init>")`
- `method_param(method_id`$_1$`, 0, `τ_5`)`
- `method_param(method_id`$_1$`, 1, `τ_6`)`
- ~~`supertype(`τ_5`, `τ_2`)`~~
- `supertype(`τ_6`, `τ_3`)`

We can see that now the remaining constraints are satisfiable with the original facts above in §5.2.1. Furthermore, as `com.lambdaworks.redis...Intent` does not accept `jdk:java.lang.Class` as a second argument, it cannot satisfy the above constraints. Thus, we can precisely determine using the constraints that `Intent` is from `Android` but not `Redis`.

To accomplish our goal and ignore `Main`, Scitix assigns the `Any` type to TVs with class and method constraints where the TVs do not have a matching type in the knowledge base. Scitix generates the corresponding fact for this type, `class("any", "any", "Main")` which satisfies the constraint `class(`τ_2`, _, "Main")`. However, the `Any` type generated by Scitix does not yet satisfy the constraint `supertype(`τ_5`, `τ_2`)`. While one could potentially generate facts for every type τ_5 can potentially represent, this is impractical. The TV τ_5 may potentially match a large number of types from the knowledge base because different types may have the same simple name or different types may contain methods with the same name, especially as the knowledge base becomes larger. To maintain scalability, Scitix simply deletes all `supertype` constraints and refines the solution by adding back `supertype` constraints iteratively. As long as adding the super type constraint does not lead to the set of constraints becoming unsatisfiable, the constraint is then kept for future iterations. That way, the constraint `supertype(`τ_6`, `τ_3`)` is preserved and strongly constrains that the second argument in `Intent`'s constructor is a `Class` type. Now, Scitix can precisely determine, using constraint-based type inference, that `Intent` comes from `Android` and not `Redis`.

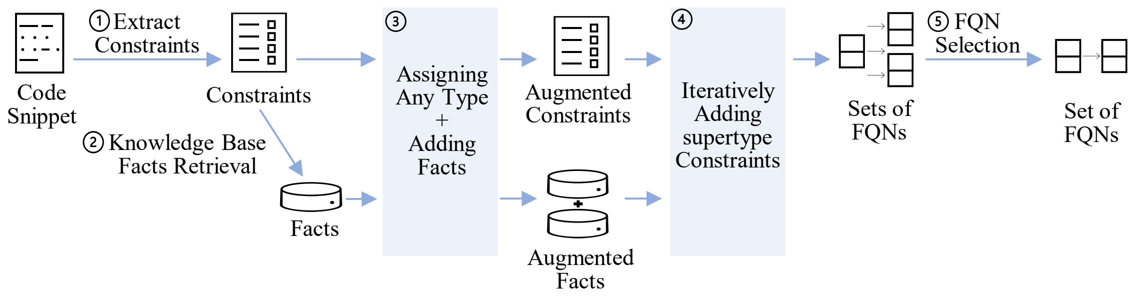


Figure 5.2: The workflow of Scitix.

5.3 Methodology

Figure 5.2 illustrates the general workflow of Scitix, which infers a set of FQNs from a given code snippet. The inferred FQNs can be used to add import statements to the code snippet to make it compilable. In step ①, constraints are extracted from the code snippet (§5.3.1). In step ②, relevant facts are retrieved from the knowledge base (§5.3.2). During step ③, Scitix assigns `Any` type to the types in the code snippet that can be identified as unknown types, and adds the required facts after the assignment (§5.3.3). To efficiently handle unknown types that cannot be identified, instead of performing constraint solving with all the constraints at the beginning, Scitix starts with a satisfiable subset of constraints, and expands this set by iteratively adding constraints that preserve satisfiability (step ④, §5.3.4). This iterative process continues until either all constraints have been evaluated or a timeout is reached. Finally, sets of FQNs that satisfy the constraints are returned. One such set is selected (step ⑤, §5.3.6) and used to add missing import statements and libraries to the code snippet.

5.3.1 Extracting Constraints

Table 5.2 shows the constraints used by Scitix which are simplified from the original constraints in §3. To extract the initial constraints from the AST, Scitix employs SnR, which performs static analysis based on the simplified rules outlined in Table 5.3. The `paramtype` constraints are refined by SnR using the process described in §3.3.4, thus allowing Scitix to handle parameterized types as well.

Table 5.2: Descriptions of constraints on type variables used by Scitix.

Name	Arguments	Description
class	(τ, `FQN`, `simple name`)	τ has the given `FQN` and `simple name`.
supertype	(τ_1, τ_2)	τ_1 is the super type of τ_2.
method	(`id`, τ, τ_{return}, `method name`)	τ has a method with the given `method name` which returns τ_{return} and is identified by `id`.
method_param	(`id`, `arg_number`, τ)	method with `id` takes in a parameter at index `arg_number` with type τ.

5.3.2 Retrieving Facts From the Knowledge Base

Next, all the known facts for FQNs that the types in the code snippet might represent are extracted from the knowledge base and given to Scitix. The knowledge base is built in the same manner as SnR detailed in §3.3.1. Table 5.4 provides a brief summary of the various kinds of facts stored in the knowledge base.

5.3.3 Assigning the `Any` Type and Adding Facts

As mentioned in §5.2, there may be no type in the knowledge base that corresponds to the type represented in the code snippet. (*i.e.*, no type in the knowledge base satisfies the `class`, `method`, and `method_param` constraints extracted for the type variable identified in the snippet). Such a type variable can be directly identified as an unknown type. To handle these identified unknown types, Scitix assigns the `Any` type to each of them. After the assignment, new facts that make the `Any` type satisfy the originally unsatisfiable `class`, `method`, and `method_param` constraints need to be generated and added to the original set of facts gathered by SnR. Table 5.5 shows an example of the generated facts after an `Any` type assignment. As shown in the table, the original constraints require τ to have a certain simple name and a certain method with certain arguments and a return type. To make the `Any` type satisfy these constraints, facts that state the `Any` type has the same simple name and the same method are generated. To allow the `Any` type and the return type of the method to be resolved independently, Scitix also augments the method constraint to decouple these two types. The augmented method constraint is also shown in Table 5.5.

Compared to `supertype` constraints, `class`, `method`, and `method_param` constraints can be directly queried from the knowledge base. Therefore, they can be solved more efficiently. When no solution can be found in the knowledge base, it indicates that a type is unknown.

Table 5.3: Simplified key statements and expressions used to extract constraints from code snippets. Type name can either be a simple name or a qualified name which is denoted as t in expressions. $\vec{x}$ denotes a vector of x where x can be an expression e, or a statement s. $\vec{\tau_1}[i]$ represents the ith element in $\vec{\tau_1}$ where the vector index starts from 0 inclusive to the size of $\vec{\tau_1}$, denoted by $|\vec{\tau_1}|$ exclusive. e: τ_1 denotes the expression e is mapped to the type variable τ_1.

Category	Name	Code	Example	Type Variables	Generated Constraints

Class	Declaration	$cls\ c\ ext\ t_1\ impl\ \vec{t_2}$	*cls* `C` *ext* `S` *impl* `I`	$c : \tau, t_1 : \tau_1, \vec{t_2} : \vec{\tau_2}$	supertype(τ_1, τ)
					`for` i `from` 0 `to` $\lvert\vec{\tau_2}\rvert$:
					  supertype($\vec{\tau_2}[i]$, τ)

Type Name	Simple Name	sn	`View`	sn: τ_1	class(τ_1, _, sn)
	Qualified Name	$n\ .\ sn$	`android.view.View`	$n\ .\ sn$: τ_1	class(τ_1, $n\ .\ sn$, sn)
Statement	If	if (e) $\{\vec{s}\}$	`if (true) {}`	e: τ_1	class(τ_1, "boolean", "boolean")
	While	while (e) $\{\vec{s}\}$	`while (true) {}`	e: τ_1	class(τ_1, "boolean", "boolean")
Expression	Assignment	$e_1\ =\ e_2$	`a = 12`	e_1: τ_1, e_2: τ_2	supertype(τ_1, τ_2)
	Annotation	@ t	`@Override`	t: τ_1	class(τ_1, _, t)
	Declaration	$t\ i$	`View view`	t: τ_1	class(τ_1, _, t)

	Method	$e_1\ .\ m(\vec{e_2})$	`string.substring(1)`	e_1: τ_1, $\vec{e_2}$: $\vec{\tau_2}$, $e_1.m(\vec{e_2})$: τ_3
				[create τ_p for τ_s in $\vec{\tau_2}$] $= \vec{\tau_4}$
					method(mid, τ_1, τ_3, m)
					`for` i `from` 0 `to` $\lvert\vec{\tau_2}\rvert$:
					  method_param(mid,i,$\vec{\tau_4}[i]$)
					  supertype($\vec{\tau_4}[i]$, $\vec{\tau_2}[i]$)

	New Instance	new $t(\vec{e})$	`new String()`	t: τ_1, $\vec{e}$: $\vec{\tau_2}$, new $t(\vec{e})$: τ_1
				[create τ_p for τ_s in $\vec{\tau_2}$] $= \vec{\tau_4}$
					method(mid, τ_1, τ_1, "<init>")
					`for` i `from` 0 `to` $\lvert\vec{\tau_2}\rvert$:
					  method_param(mid,i,$\vec{\tau_4}[i]$)
					  supertype($\vec{\tau_4}[i]$, $\vec{\tau_2}[i]$)

	Array Access	$e_1[e_2]$	`a[1]`	e_2: τ_1	class(τ_1, "int", "int")

Leveraging this insight, Scitix efficiently detects some unknown types and assigns them with the `Any` type which better models them during type inference.

5.3.4 Iteratively Adding Supertype Constraints

Although some unknown types can be identified and handled in the last step, there can still be some unknown types remaining, rendering the entire set of constraints unsatisfiable. To tackle this, Scitix utilizes an iterative constraint-solving approach to efficiently search for a maximal set of satisfiable constraints. Specifically, Scitix adds `supertype` constraints one-by-one to the previously satisfied `class`, `method`, and `method_param` constraints, and ensures the satisfiability of the maintained set of constraints. As illustrated in Algorithm 4, the algorithm proceeds greedily. During each iteration, a `supertype` constraint s is added to the currently satisfiable set C. If the resulting $C \cup \{s\}$ is satisfiable, s is retained, and the process continues with the updated set $C = C \cup \{s\}$. Otherwise, s is discarded and excluded from future iterations. The final satisfiable set of constraints is then used to query for the mapping between the TVs and the types.

Table 5.4: Description of the class, super type, method, and method parameter facts including their respective attributes stored in a knowledge base. The parameterized type information on types is omitted from the facts for clarity.

Fact Type	Attributes	Description
`class`	(`Type`, `FQN`, `sn`)	`Type` has the given `FQN` and simple name (`sn`).
`supertype`	($\texttt{Type}_1$, $\texttt{Type}_2$)	$\texttt{Type}_1$ is super type of $\texttt{Type}_2$.
`method`	(`id`, `Type`, $\texttt{Type}_r$, `name`)	`Type` has a method with the given `name` which returns Type_r and uniquely generated `id`.
`method_param`	(`id`, `arg_number`, `Type`)	method with `id` takes in a parameter at index `arg_number` of the type, `Type`.

Table 5.5: Augmentations to the existing constraints along with additional generated facts to assign `TV` to the `Any` type. The generated facts contain the constants found in the original constraints. Any variables that remain in the generated facts (*e.g.* `id` in method and method parameter constraints, `FQN` or potentially `simple name`) are replaced by generated constants.

Name	Constraint Arguments	Augmented Constraint Arguments	Generated Fact Arguments
`class`	(τ, `FQN`, `simple name`)	-	(`Any`, `FQN`, `simple name`)
`method`	(`id`, τ, τ_{return}, `method name`)	(`id`, τ, _, `method name`)	(`id`, `Any`, `Any`, `method name`)
`method_param`	(`id`, `arg_number`, `TV`)	-	(`id`, `arg_number`, `Any`)

This approach contrasts with the method described in §5.3.3 where `Any` types facts were added to handle unknown types. This is because adding `Any` type facts is not practical for `supertype` constraints since `Any` is both super and subtype of all the types. Adding facts for every type in the knowledge base slows down constraint solving and would not be scalable as the number of types in the knowledge base grows. To maintain scalability in the presence of unknown types, Scitix performs only a single iteration through the set of `supertype` constraints. This is sufficient to discover a locally maximal set of satisfiable constraints. A key insight is that if a set of constraints C contains an unsatisfiable subset of constraints C_s, then C itself is also not satisfiable. Leveraging this property, the algorithm incrementally builds up the constraint set from a known satisfiable subset. Although this greedy strategy does not guarantee a globally maximal solution, it scales efficiently, avoiding the exponential complexity of exhaustive global search, and reliably produces a subset that is sufficient in practice. After running Algorithm 4, the unsatisfiable `supertype` constraints that arise from unknown types and cannot be precisely identified in §5.3.3 are now decoupled from the constraint-solving process.

Algorithm 4: Iteratively adding `supertype` constraints.

```
   Input  : $C$, the set of gathered class, method, and method_param constraints.
   Input  : $S$, the set of gathered supertype constraints.
   Input  : $F$, the set of gathered facts.
   Output: The extended set of constraints including the satisfiable supertype
           constraints.
1  foreach $s \in S$ do
2      if satisfiable($C \cup \{s\}$, $F$) then // Check satisfiability of the constraint set with
        the new constraint
3          $C \leftarrow C \cup \{s\}$
4  return $C$
```

5.3.5 Generating the Query

At this stage, the previously collected TVs and facts can be used to identify types that satisfy the queried constraints. To accelerate this process, a couple of strategies are employed to guide the solver and filter the input effectively.

Constraint Ordering. To guide the constraint solver, Soufflé [20], constraints are ordered to determine which should be computed first for the final solution. While the order of the constraints does not affect the result, significant speed differences can stem from the bottom-up evaluation strategy used by Soufflé. Recent work has explored automatically optimizing the ordering [114]. However, using the automatic optimizer would require profiling the query for each code snippet, negating any time savings. Instead, a simple but effective strategy is applied to order the constraints by their ease of evaluation, *i.e.*, `class`, `method`, `method_param`, and `supertype`. The `class` constraints are straightforward to evaluate, requiring iterating through class facts and matching them to the simple or fully qualified name given in the constraint. This can be quickly accomplished. Moreover, subsequent constraints on the same TV only need to iterate through types with the given simple name, significantly reducing the search space. The `method` constraint is similar to the `class` constraint, but since there are more method facts than class facts, `method`, and `method_param` constraints were evaluated after the `class` constraints. Conversely, `supertype` constraints are time-consuming as they generally require iterating through all of the `supertype` facts. Therefore, `supertype` constraints were handled last.

Library Filtering. To prevent the size of the knowledge base from overwhelming the solver, only a subset of relevant libraries is selected as input. We observe that *code snippets typically reference libraries using the simple names of the types used.* Leveraging this observation, a subset of libraries is extracted from the knowledge base, containing only those that define a type with a simple name appearing in the code snippet. All queries to the knowledge base are then restricted to this subset, significantly reducing the search space and improving Scitix's scalability.

In contrast, `SnR` cannot filter libraries using only simple names. Code snippets commonly use variables without declaration and thus without simple names (*e.g.*, `timepicker` in Figure 5.1). Moreover, method calls can return types without explicitly mentioning a simple name, even from other libraries (transient dependency). Since `SnR` cannot handle unknown types and unsatisfiable constraints well, it considers all libraries using both simple names and method names, at the cost of scalability.

5.3.6 FQN Selection

After queries to the constraint solver return a set of unique mappings between TVs and types, a single best-suited mapping is selected using a simple heuristic. This heuristic considers the types that require import statements. While prior work focused solely on minimizing the number of libraries used, that heuristic is extended to identify the most complete solution from our constraint optimization results and to address cases where multiple solutions involve the same minimal number of libraries. The final mapping is selected based on the following ordering criteria: 1. Fewest `Any` type, 2. Fewest number of libraries, 3. Highest libraries score, and 4. Fewest number of prefixes.

Least `Any` Type. The solution with the least `Any` type that satisfies the constraints will maximize the recovery of FQNs in the code snippet. While prior work considers the least number of libraries first, this does not make sense for Scitix as not recommending any FQN would technically use the fewest libraries. Thus, Scitix filters for the least `Any` types first.

Least Number of Libraries. Prior work has solely used the least number of libraries effectively as the selection heuristic [4]. In recognition of this, the same approach was adopted here.

Highest Libraries Score. The library score was used to differentiate between solutions that have the *same* minimal, total number of libraries. This scenario arises when multiple libraries implement multiple types with the same simple name, but no single library implements all the required types. For example, if eight required types come from two libraries, a solution that derives seven types from one library and only one from the other is preferred over a solution that evenly splits the types across both libraries. While both libraries may contribute necessary functionalities, the library containing a more complete set of required types is prioritized, with the second library included only for the specific type it uniquely provides.

To calculate the library score for n libraries, the number of types that are in each library is counted. The counts are placed in an ordered list of integers $[a_0, a_1, ..., a_{n-1}]$ where a_0 represents the smallest number of types contributed by a single library, and subsequent values correspond to increasing contributions. The library score is computed by multiplying the number of types in each library with the index, $\sum_{i=0}^{n-1} i * a_i$. Note that by this stage, all mappings contain the same number of libraries and resolve the same number of types according to the first two selection criteria.

Number of Prefixes. When multiple libraries provide the required type, the number of prefixes serves as a tiebreaker. For example, the FQN `a.b.C` has the prefixes `a` and `a.b`. The total number of unique prefixes is calculated for all types within a mapping, and mappings with fewer prefixes are preferred. This approach also helps distinguish cases where a library repackages another, as the repackaging process typically introduces additional prefixes to the original FQN.

Example. In Figure 5.1, `View`, with only a `class` constraint, matched multiple types in the knowledge base in Table 5.1. To determine which type to select, Scitix follows the list of four criteria in order. Since `View` is not assigned to `Any`, the least `Any` type rule does not apply. The least number of libraries rule narrows our selection to `android.view.View` from the `android` library and `javax.swing.text.View` from the `JDK` as these libraries are referenced by other TVs in the code snippet. Next, from the library score rule, Scitix can precisely select `android.view.View` as the type for `View`, as there are four other TVs in the code snippet with types from the `android` library, whereas only one TV in the code snippet is from the `JDK`. By following the rules, Scitix can concretely determine the correct FQNs for the types in the code snippet that match multiple types in the knowledge base.

5.4 Evaluations

Our evaluation was guided by the following research questions:

RQ1 How well does Scitix infer import statements as the knowledge base scales?
RQ2 How does Scitix's runtime scale with increasing knowledge base size?
RQ3 What is the contribution of different Scitix components to its overall precision?
RQ4 How does Scitix compare with LLMs in inferring import statements?
RQ5 How well does Scitix perform in the presence of unknown types?

Table 5.6: The number of code snippets that use a given language feature in the StatType-SO and ThaliaType datasets.

Language Features	StatType-SO	ThaliaType	Language Features	StatType-SO	ThaliaType
super	43	0	array	32	23
type cast	68	208	parameterized type	57	179
assignment	255	300	wildcard type	7	60

5.4.1 Experiment Setup

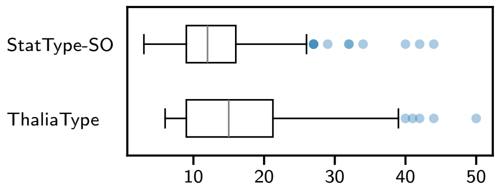
Scitix. We implement Scitix in Java. In Scitix, MariaDB served as the knowledge base; initial constraints were provided by SnR; and the constraints were solved by Soufflé Datalog solver. A replication package is available at https://figshare.com/s/f03c5103e2ab02125b83 .

Baseline Approaches. We evaluated Scitix against SnR along with state-of-the-art LLMs including GPT-4o and GPT-4o-mini [24], as well as state-of-the-art open-weight models such as Llama3.1:70b and Llama3.1:8b [25], all of which have shown strong performance in type inference for code snippets [7, 8, 5, 115].

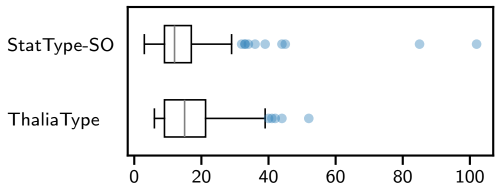
Dataset: StatType-SO. Using the same dataset as we did in §§3 and 4, we utilized *StatType-SO* [2]. StatType-SO consists of 267 manually repaired Stack Overflow code snippets from 6 popular Java libraries, including 49 jars when considering the dependencies.

Knowledge Bases. The initial knowledge base, denoted as Γ_0, was constructed using the 49 jars used in StatType-SO. To evaluate scalability, we progressively expanded the knowledge base by incorporating additional jar files. Following the procedure from prior work [78], we expanded the knowledge base by incorporating the top 500, 1,000, 1,500, 2,000, 2,500, and 3,000 most popular jars from the Maven repository. This resulted in the expanded knowledge bases Γ_{500}, Γ_{1000}, Γ_{1500}, Γ_{2000}, Γ_{2500}, and Γ_{3000}.

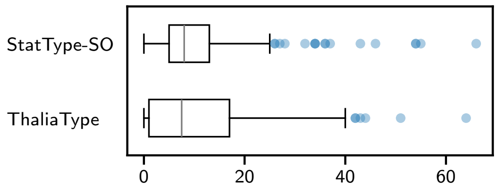
Dataset: ThaliaType. To ensure a fair evaluation and mitigate potential data leakage concerns (*i.e.*, LLMs were likely trained on code snippets from StatType-SO), we additionally used *ThaliaType* [115] introduced in §4. The code snippets were generated using the same six libraries as StatType-SO to ensure that the libraries evaluated were familiar to the LLMs. As shown earlier in Figure 4.5, the code snippets from ThaliaType and StatType-SO are comparable in length and number of import statements. Both StatType-SO and ThaliaType exercise a diverse set of language features, such as parameterized types and wildcard types (Table 5.6). While ThaliaType results in more type variables during type inference, it results in slightly fewer method and method parameter constraints compared



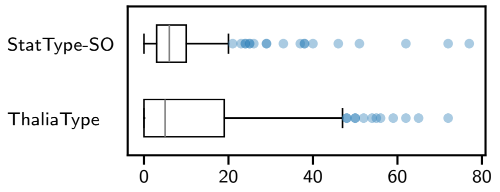
(a) Comparing the number of type variables.



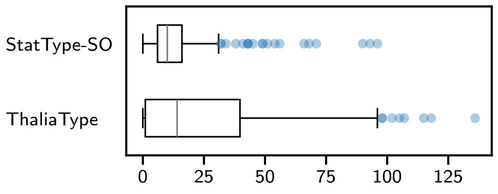
(b) Comparing the number of class constraints.



(c) Comparing the number of method constraints.



(d) Comparing the number of method parameter constraints.



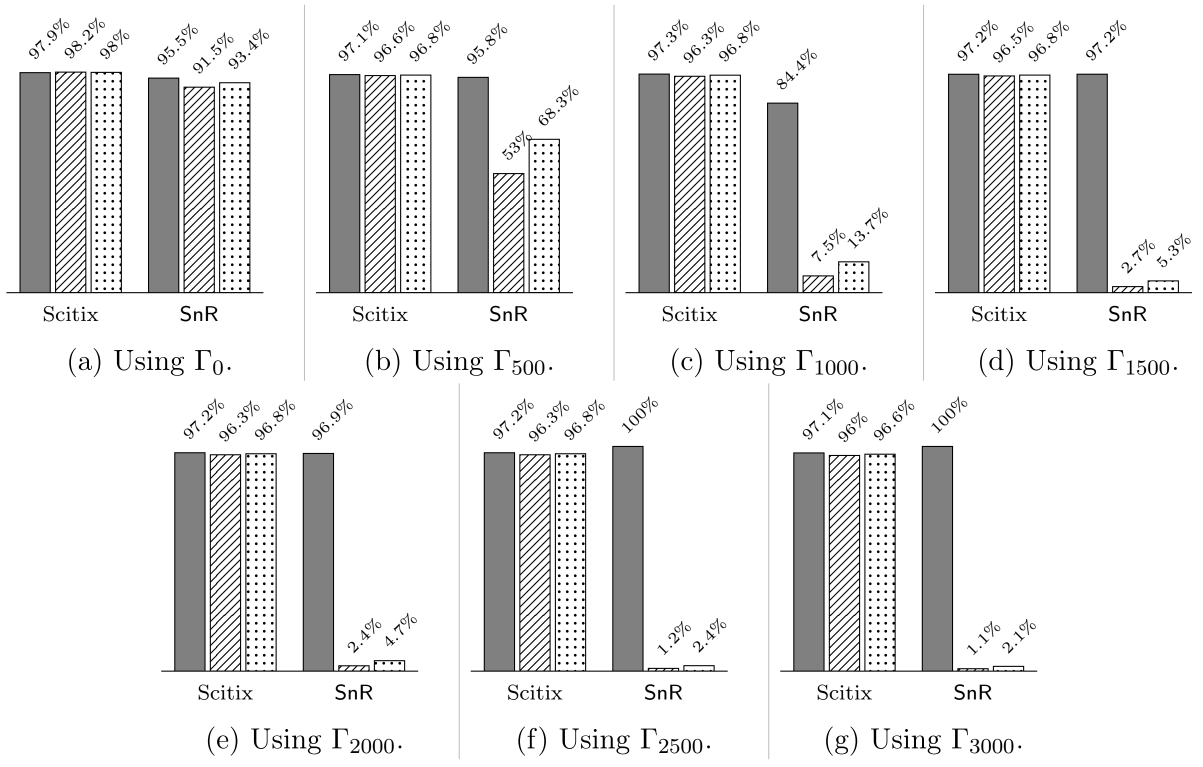
(e) Comparing the number of super type constraints.

Figure 5.3: Box plots comparing the number of type variables and constraints gathered in StatType-SO and ThaliaType.

to StatType-SO (Figure 5.3).

Configurations. All experiments were conducted on a Linux system equipped with an AMD 5600G CPU, limited to 16GB of RAM, and a five-minute timeout per code snippet. This setup emulates a typical development environment, albeit with a slightly longer timeout to better explore both tools' practicality. The Llama3.1:8b and Llama3.1:70b models were hosted and accessed via a web API backed by an RTX 6000 Ada GPU, while GPT-4o and GPT-4o-mini were accessed through the OpenAI APIs.

Evaluation Metrics. Scitix is evaluated using the same precision, recall, and F1-scores from §§3 and 4.



(a) Using Γ_0. (b) Using Γ_{500}. (c) Using Γ_{1000}. (d) Using Γ_{1500}.

(e) Using Γ_{2000}. (f) Using Γ_{2500}. (g) Using Γ_{3000}.

Figure 5.4: Comparison of Precision, Recall, and F1-score on StatType-SO. Legend: ■ Precision, ▨ Recall, ⬚ F1-score.

$$\text{Precision} = \frac{\text{Correctly Inferred FQNs}}{\text{All Inferred FQNs}} \qquad \text{Recall} = \frac{\text{Correctly Inferred FQNs}}{\text{All Expected FQNs}} \qquad \text{F1} = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$$

5.4.2 RQ1: How well does Scitix infer import statements as the knowledge base scales?

We evaluated Scitix's performance on code snippets from the StatType-SO and ThaliaType datasets without import statements. Across all knowledge base sizes, Scitix consistently outperformed SnR, particularly as the knowledge base size increased, demonstrating superior scalability. As shown in Figures 5.4 and 5.5, Scitix reduced the percent of errors ($\frac{\text{incorrectly inferred}}{\text{all expected}}$) by 79% on StatType-SO and 37% on ThaliaType, relative to SnR, even on the smallest knowledge base (Γ_0). Moreover, Scitix achieved F1-scores of 98.0% (StatType-SO) and 91.1% (ThaliaType), compared to SnR's 93.4% and 84.3% on the original Γ_0 knowledge base.

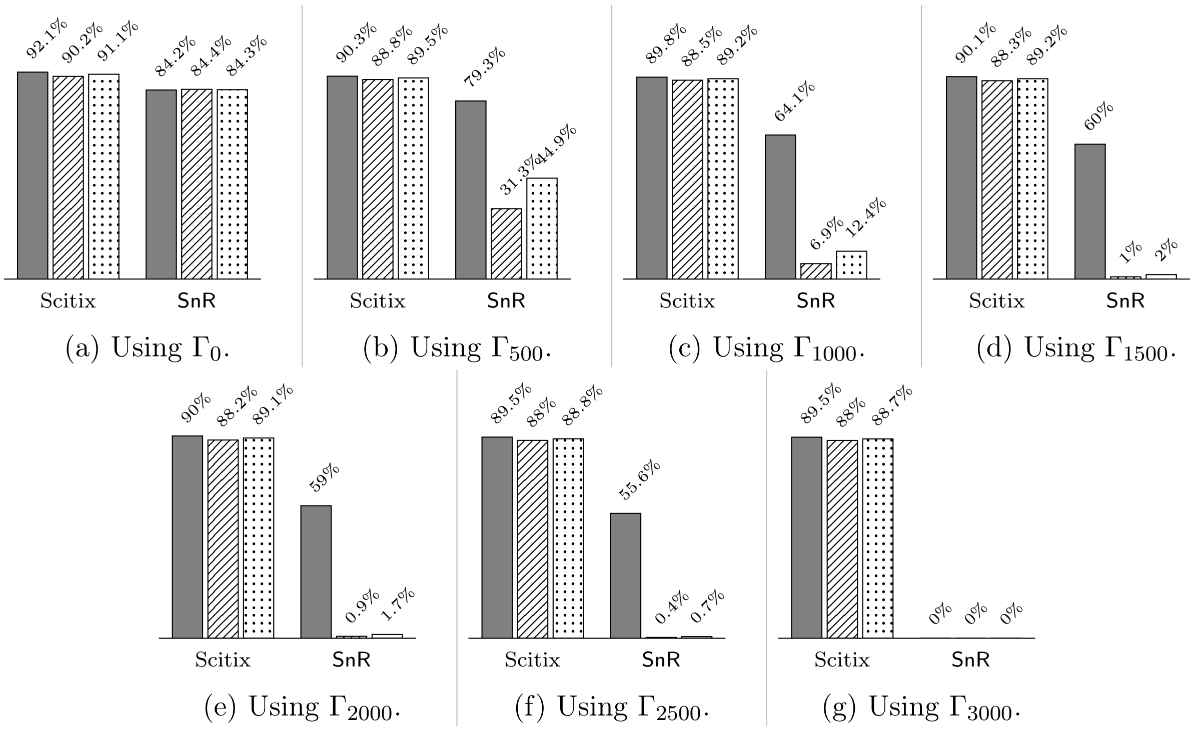
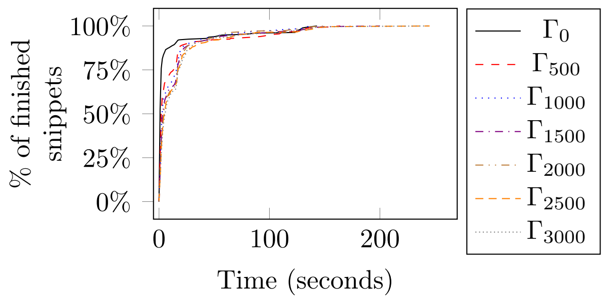


Figure 5.5: Comparison of Precision, Recall, and F1-score on ThaliaType. Legend: ■ Precision, ▨ Recall, ⬚ F1-score.

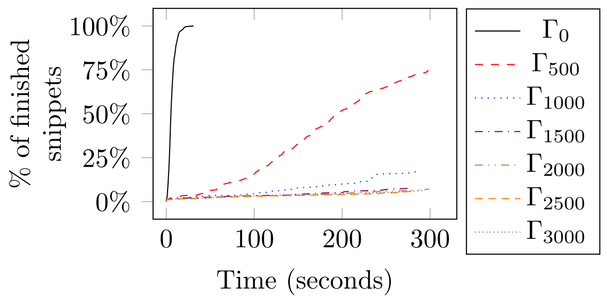
As the knowledge base expanded, the gap between Scitix and SnR widened, with SnR increasingly failing to complete within the five-minute timeout per code snippet. At the largest knowledge base size (Γ_{3000}), Scitix achieved F1-scores of 96.6% (StatType-SO) and 88.7% (ThaliaType), whereas SnR dropped to 2.1% and 0.0%, respectively. Scitix's inference performance remained consistent as the knowledge base size grew, demonstrating the scalability of our approach.

Although SnR did not experience timeouts on the Γ_0 knowledge base (shown in RQ2), its inadequate handling of unknown types, such as user-defined classes and wildcard types, led to a decrease in performance. Additionally, Scitix's enhanced heuristics contributed to higher precision and recall, even on Γ_0 knowledge base. For this benchmark, all libraries used by the code snippets in StatType-SO are included in the knowledge base. For real-world use, we should strive to build as large of a knowledge base as possible to ensure high recall for any code snippet.

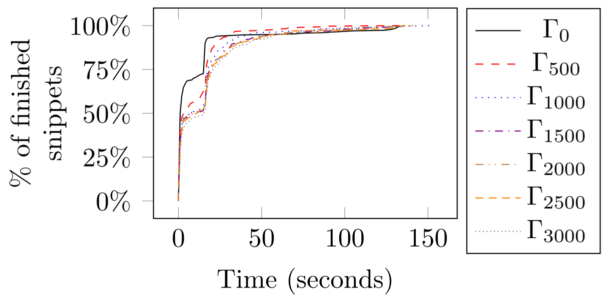
Scitix performs worse on ThaliaType than on StatType-SO because there is less in-



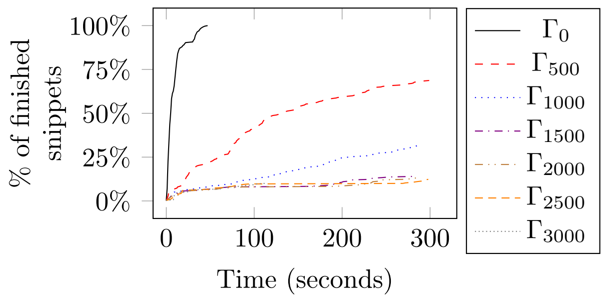
(a) Scitix on StatType-SO code snippets using various knowledge bases.



(b) SnR on StatType-SO code snippets using various knowledge bases.



(c) Scitix on ThaliaType code snippets using various knowledge bases.



(d) SnR on ThaliaType code snippets using various knowledge bases.

Figure 5.6: Graph of Scitix and SnR's running time. The Y-axis shows the accumulated percentage of finished code snippets for a given time on the X-axis.

formation, such as methods, to help narrow down the potential solutions (Figure 5.3). Simultaneously, there are more type variables to solve for in ThaliaType. Consequently, with fewer constraints for each type, there is often not enough information to decide between different FQNs. This affected both Scitix and SnR.

5.4.3 RQ2: How does Scitix's runtime scale with increasing knowledge base size?

To evaluate scalability, we analyzed how Scitix's runtime scaled with increasing knowledge base size and compared it to SnR. Figure 5.6 illustrates the time taken for Scitix and SnR to complete type inference on StatType-SO and ThaliaType code snippets using Γ_0, Γ_{500}, Γ_{1000}, Γ_{1500}, Γ_{2000}, Γ_{2500}, and Γ_{3000} knowledge bases. The Y-axis represents the percentage

Table 5.7: The precision, recall, and F1-scores completed by Scitix, Scitix $_{\text{Simple}}$, Scitix $_{\text{Random}}$, Scitix $_{\text{Filter}}$, and Scitix $_{\text{Naive}}$ using Γ_0, Γ_{500}, Γ_{1000}, Γ_{1500}, Γ_{2000}, Γ_{2500}, and Γ_{3000} knowledge bases. The best result in each column is shown in bold.

		Γ_0			Γ_{500}			Γ_{1000}			Γ_{1500}			Γ_{2000}			Γ_{2500}			Γ_{3000}		
	Variant	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
StatType-SO	Scitix	**97.9%**	**98.2%**	**98.0%**	**97.1%**	**96.6%**	**96.8%**	**97.3%**	**96.3%**	**96.8%**	**97.2%**	**96.5%**	**96.8%**	**97.2%**	**96.3%**	**96.7%**	**97.2%**	**96.3%**	**96.7%**	**97.1%**	**96.0%**	**96.6%**
	Scitix $_{\text{Simple}}$	97.3%	97.2%	97.3%	96.6%	95.8%	96.2%	96.7%	95.3%	96.0%	96.5%	95.1%	95.8%	96.5%	95.1%	95.8%	96.6%	95.1%	95.8%	96.3%	95.1%	95.7%
	Scitix $_{\text{Random}}$	97.3%	**98.2%**	97.7%	96.4%	96.4%	96.4%	96.7%	96.1%	96.4%	96.7%	96.0%	96.3%	96.7%	96.0%	96.3%	96.7%	**96.3%**	96.5%	96.3%	95.9%	96.1%
	Scitix $_{\text{Filter}}$	97.6%	**98.2%**	97.9%	97.0%	**96.6%**	**96.8%**	97.2%	**96.3%**	96.7%	97.1%	95.2%	96.1%	97.1%	94.5%	95.8%	97.1%	94.5%	95.7%	97.0%	94.2%	95.6%
	Scitix $_{\text{Naive}}$	96.4%	94.8%	95.6%	95.6%	88.3%	91.8%	94.6%	87.0%	90.6%	96.0%	86.0%	90.7%	96.8%	85.4%	90.8%	96.5%	85.4%	90.6%	95.7%	86.0%	90.6%
ThaliaType	Scitix	92.1%	**90.2%**	91.1%	**90.3%**	**88.8%**	**89.5%**	**89.8%**	**88.5%**	**89.2%**	90.1%	**88.3%**	**89.2%**	90.0%	88.2%	**89.1%**	89.5%	88.0%	**88.8%**	**89.5%**	88.0%	**88.7%**
	Scitix $_{\text{Simple}}$	91.5%	88.9%	90.2%	89.8%	87.3%	88.6%	89.5%	87.1%	88.3%	89.6%	87.0%	88.3%	89.6%	87.0%	88.3%	89.0%	86.8%	87.9%	89.0%	86.8%	87.9%
	Scitix $_{\text{Random}}$	91.2%	90.0%	90.6%	90.0%	88.7%	89.3%	89.4%	**88.5%**	88.9%	89.7%	**88.3%**	89.0%	89.6%	**88.3%**	89.0%	89.2%	**88.1%**	88.7%	89.0%	**88.1%**	88.6%
	Scitix $_{\text{Filter}}$	**92.2%**	**90.2%**	**91.2%**	90.2%	**88.8%**	**89.5%**	**89.8%**	**88.5%**	89.1%	90.0%	**88.3%**	89.1%	90.0%	88.2%	**89.1%**	89.4%	**88.1%**	88.7%	89.4%	88.0%	**88.7%**
	Scitix $_{\text{Naive}}$	89.4%	84.5%	86.8%	90.8%	76.4%	83.0%	89.2%	74.6%	81.2%	**94.5%**	72.5%	82.1%	**94.3%**	70.7%	80.8%	**96.3%**	70.8%	81.6%	88.4%	70.4%	78.4%

Table 5.8: The average running time in seconds with Scitix, Scitix $_{\text{Simple}}$, Scitix $_{\text{Random}}$, Scitix $_{\text{Filter}}$, and Scitix $_{\text{Naive}}$ using Γ_0, Γ_{500}, Γ_{1000}, Γ_{1500}, Γ_{2000}, Γ_{2500}, and Γ_{3000} knowledge bases.

	StatType-SO							ThaliaType						
Tool	Γ_0	Γ_{500}	Γ_{1000}	Γ_{1500}	Γ_{2000}	Γ_{2500}	Γ_{3000}	Γ_0	Γ_{500}	Γ_{1000}	Γ_{1500}	Γ_{2000}	Γ_{2500}	Γ_{3000}
Scitix	10	14	13	15	15	17	17	11	11	14	16	17	17	17
Scitix $_{\text{Simple}}$	2	5	6	7	7	8	8	4	8	10	11	12	13	13
Scitix $_{\text{Random}}$	18	23	25	28	25	28	27	21	17	20	21	20	21	23
Scitix $_{\text{Filter}}$	9	15	16	19	20	20	21	11	13	15	16	18	18	18
Scitix $_{\text{Naive}}$	9	15	13	15	15	17	17	11	11	14	16	17	17	17

of code snippets that finished within the time shown in the X-axis.

Scitix demonstrated strong scalability. With Γ_{3000}, Scitix completed all code snippets within 243 seconds. Most code snippets using Scitix finished quickly. Half of the StatType-SO code snippets finished within 6 seconds, with only eight snippets exceeding two minutes. On Γ_0, Scitix took an average of 10 seconds (StatType-SO) and 11 seconds (ThaliaType). Scitix scaled efficiently, with only a slight increase in runtime on Γ_{3000}, taking an average of 17 seconds for both StatType-SO and ThaliaType.

In contrast, while SnR performed well on Γ_0 (average of 7 seconds and 9 seconds on StatType-SO and ThaliaType, respectively), SnR failed to scale. On Γ_{3000}, SnR averaged 289 seconds (StatType-SO) and 274 seconds (ThaliaType), failing to complete inference for 93% of the StatType-SO code snippets within the five-minute timeout. Notably, Scitix did not timeout on any code snippets in StatType-SO and ThaliaType.

5.4.4 RQ3: What is the contribution of different Scitix components to its overall precision?

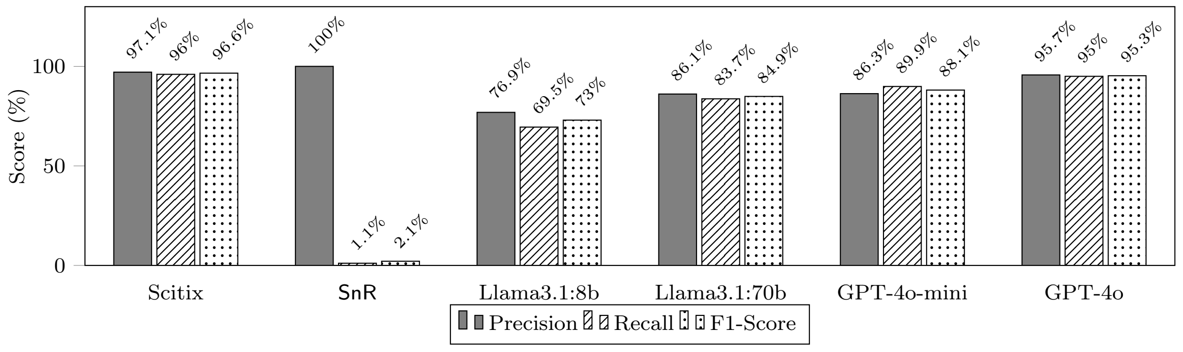
We conducted an ablation study to assess the contribution of different components of Scitix to its overall precision. By systematically disabling individual features while keeping all other components fixed, we created four variants. Scitix $_{\text{Simple}}$ omits super type constraints. Scitix $_{\text{Random}}$ randomizes the order of constraints passed to the constraint solver. Scitix $_{\text{Filter}}$ disables library filtering. Scitix $_{\text{Naive}}$ simplifies the final FQN selection by choosing the solution with the least number of any types. These four variants were evaluated on both the StatType-SO and ThaliaType datasets, and the results are summarized in Table 5.7.

Our findings indicate that all components contribute to Scitix's overall performance. Scitix consistently achieved the highest F1-scores across all knowledge bases on both datasets, with the sole exception of Scitix $_{\text{Filter}}$ on Γ_0 in the ThaliaType dataset, which was 0.1% lower. While package filtering often does not affect inference performance in ThaliaType, it did improve Scitix's runtime (Table 5.8). Scitix $_{\text{Filter}}$ without filtering was 19% slower on StatType-SO and 6% slower on ThaliaType, taking an average of 21 and 18 seconds for inference, respectively, compared to Scitix with package filtering.

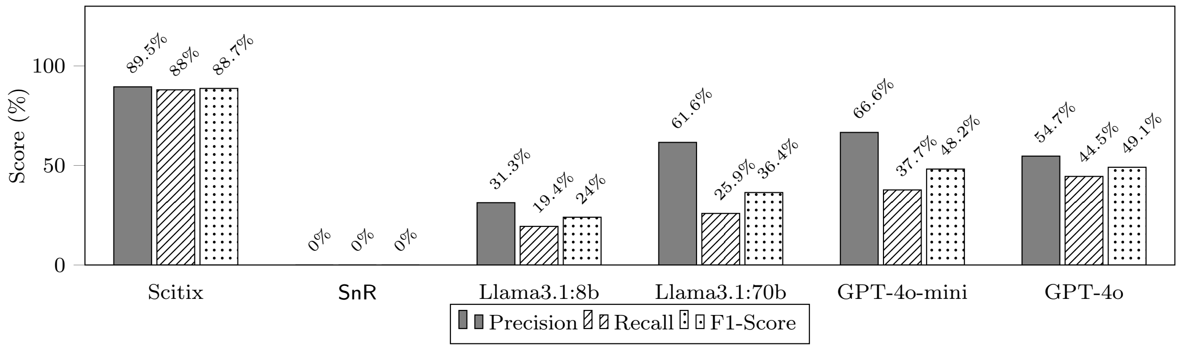
5.4.5 RQ4: How does Scitix compare with LLMs in inferring import statements?

Given the widespread use of LLMs, we compared against four state-of-the-art LLMs for the task of type inference. However, LLMs are typically trained on vast datasets scraped from the internet, which likely include code snippets from platforms like Stack Overflow. Since StatType-SO is composed of real-world Stack Overflow code snippets and has been publicly available on GitHub since 2018, there is a risk that LLMs may have been trained on this data [115]. This raises concerns regarding *data leakage*, where the benchmark data could overlap with the model's training set. Thus as a result, LLMs' performance on StatType-SO is likely overstated, whereas performance on ThaliaType, which is composed entirely of unseen code snippets, mitigates this risk.

To ensure a comprehensive and fair evaluation, we used both the StatType-SO and ThaliaType dataset, and utilized a prompt from the previous chapter (Figure 4.2a). Each code snippet was given to LLMs without import statements, and the inferred import statements were collected. Following best practice [50], we evaluated both state-of-the-art closed-source models, GPT-4o and GPT-4o-mini, and open-weight models, Llama3.1:8b and Llama3.1:70b.



(a) Type inference performance on StatType-SO.



(b) Type inference performance on ThaliaType.

Figure 5.7: Precision, recall, and F1-scores of Scitix, SnR on StatType-SO and ThaliaType code snippets using the largest Γ_{3000} knowledge base compared to state of the art LLMs, Llama3.1:8b, Llama3.1:70b, GPT-4o-mini, and GPT-4o.

Scitix outperformed all LLMs on StatType-SO, despite the potential for data leakage, and greatly outperformed all models on ThaliaType (Figure 5.7). Scitix reduced the percentage of errors by 20% on StatType-SO and 78% on ThaliaType compared to the best-performing LLM (GPT-4o). On unseen code snippets in ThaliaType, Scitix outperformed GPT-4o by 80.7% in F1-score. Scitix's strong performance demonstrates that careful analysis and satisfying constraints can surpass the statistical approaches used by LLMs.

Table 5.9: The precision, recall, and F1-scores by Scitix, and SnR using Γ_0, and Γ_R knowledge bases. The best result in each column is bolded.

	StatType-SO						ThaliaType					
	Γ_0			Γ_R			Γ_0			Γ_R		
Tool	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Scitix	**97.8%**	**98.2%**	**98.0%**	**94.1%**	**97.7%**	**95.9%**	**94.8%**	**87.0%**	**90.7%**	**93.5%**	**85.6%**	**89.4%**
SnR	94.8%	91.9%	93.3%	92.6%	78.9%	85.2%	93.9%	83.2%	88.2%	92.2%	68.1%	78.3%

5.4.6 RQ5: How well does Scitix perform in the presence of unknown types?

Scitix is primarily designed for scalability (*i.e.*, incorporating significantly more types into the knowledge base), by accounting for unknown types. This design, in turn, enhances its resilience to unknown types, leading to better performance than SnR when handling code snippets that reference types not present in the knowledge base. To simulate such conditions, a smaller Γ_R knowledge base was constructed. The Γ_R knowledge base was built by selectively removing types from the original Γ_0 knowledge base. Specifically, only types that were not referenced by any other type were removed, while types that were referred to by other types were retained. This approach was chosen because, in practice, if a type is included in the knowledge base, its dependencies are typically included as well. To avoid breaking these dependencies and to ensure the setup remained realistic, types referenced by another type were retained in Γ_R. For example, if type τ_1 is a supertype of τ_2 (*i.e.*, τ_1 is referenced by τ_2), then τ_1 is retained in Γ_R. Type τ_2 is removed only if it is not referenced by any other type in the knowledge base.

After removing all unreferenced types, the number of classes decreased from 33,150 in Γ_0 to 20,732 in Γ_R. To evaluate Scitix and SnR on Γ_R, only FQNs that were still present in Γ_R were considered. As a result, the number of expected import statements in the code snippets decreased. In StatType-SO, the number dropped from 1,298 to 1,066, and in ThaliaType, from 2,685 to 1,742.

Table 5.9 shows that Scitix was largely unaffected by the additional unknown types, with only a 2.1% drop in F1-score on StatType-SO and a 1.4% drop on ThaliaType. In contrast, SnR experienced a larger decrease, with F1-scores dropping by 8.7% on StatType-SO and 11.2% on ThaliaType when using the Γ_R knowledge base. These results suggest that Scitix is more resilient to incomplete knowledge bases than SnR, making it better suited for real-world scenarios where type information may be incomplete.

5.5 Discussion

In this section, we discuss the limitations of Scitix (§5.5.1), and examine some potential threats to validity in §5.5.2.

5.5.1 Limitation

Scitix uses SnR to generate constraints. However, SnR does not support certain language features, such as wildcard types, which may lower type inference precision. Scitix mitigates these gaps by assigning unknown types from unsupported language features as `Any`. Our evaluation using StatType-SO and ThaliaType, which included these unsupported language features, demonstrated that Scitix achieved high precision despite these limitations, significantly outperforming SnR. Additionally, Scitix's contributions are orthogonal to the constraint extraction process; future advances in constraint generation will directly benefit Scitix.

5.5.2 Threats to Validity

Internal. To mitigate potential threats to validity, we investigated cases where Scitix underperformed to ensure our implementation was accurate. Evaluation subjects were executed using external scripts to maintain consistent and reproducible settings. The implementation, knowledge bases, and experiment setup are included in our replication package `https://figshare.com/s/f03c5103e2ab02125b83`.

External. One potential external threat to Scitix is its ability to generalize to other programming languages. However, our approach is not specific to Java, as concepts such as methods and inheritance are widely used in object-oriented programming languages. Scitix can be easily adapted to support other languages by using a suitable constraint generator and knowledge base.

5.6 Related Work

In this section, we introduce some related work in gradual typing, type inference for Java code snippets.

Gradual Typing. Gradual typing [112] is a popular type system that allows types to be unknown statically, similar to encountering unknown types in a code snippet. Notably, gradual typed languages, *i.e.*, TypeScript [116], and Python with mypy [117], use `Any` type to escape from type checking to allow for partially typed code. However, type checking differs from type inference in key ways. Type checking takes the types and expressions from a program to verify if the type rules are correctly followed. Scitix conducts type inference using the type rules to compute the types on code snippets. We also applied Scitix to Java, a language that does not employ gradual typing.

Type Inference on Java Code Snippets. Compared to the related works initially discussed in §3.6 and §4.8, Scitix both achieves higher performance and requires less computational resources than state-of-the-art ML-based techniques [5, 7, 8]; especially when considering the training and fine-tuning costs as well. The state-of-the-art ML-based technique [5] requires hardware (*i.e.*, a RTX 3090) that many developers may lack. Similarly, ZS4C [7], an LLM-based technique, requires data center scale compute for ChatGPT, yet still does not outperform Scitix on the StatType-SO dataset. Additionally, as explored in §4, LLM-based techniques are susceptible to *data leakage*, a common issue in engineering research [115, 50, 31, 33, 46, 103, 106, 107], potentially inflating LLM-based techniques' performance Compared to ML-based techniques, Scitix is more explainable, with rules to dictate the types to import. Furthermore, Scitix can further enhance the hybrid technique proposed by Chen et al. [8], which integrates constraint-based type inference (*i.e.*, SnR) with ML-based techniques for type inference. Scitix can improve the initial type inference, which helps refine ML predictions.

Reusing real-world Stack Overflow code snippets is challenging. Many studies have worked on large-scale repair and reuse [53, 78, 118]. CSNIPPEX [53] introduced a pipeline to convert Stack Overflow posts into compilable code units using a simple type inference scheme. It resolved dependencies for over 237,000 posts with the top 3,000 jars. APIzation [78] extended CSNIPPEX to create reusable APIs from Stack Overflow code. Zerouali et al. [118] analyzed the versions of the libraries used in Stack Overflow Java code snippets by matching class names and methods. Even after filtering out ambiguous types, they identified 435 unique Jars across 1,760 code snippets. Scitix has the potential to improve classes and methods identification, thus providing more semantic information for existing Stack Overflow Java code snippets.

5.7 Chapter Conclusion

This chapter presented Scitix, a novel approach for precise, and scalable constraint-based type inference on code snippets. Scitix is practical for real-world deployment since it maintains high precision and recall with large knowledge bases and unknown types. Scitix marks explicitly referenced types in a code snippet that are unknown to the knowledge base as a special `Any` type and iteratively adds constraints to enhance precision. We conducted a comprehensive evaluation using real-world code snippets from StatType-SO along with ThaliaType, a benchmark designed to mitigate data leakage concerns. Scitix reduced the error rate by 79% on StatType-SO and 37% on ThaliaType using Γ_0, compared to SnR. Scitix demonstrated great scalability, achieving F1-scores of 96.6% on StatType-SO and 88.7% on ThaliaType with the largest knowledge base (Γ_{3000}), while SnR only reached 2.1% and 0.0%, respectively. Furthermore, Scitix outperformed LLMs, reducing the error rate by 20% on StatType-SO and 78% on ThaliaType compared to GPT-4o, even with the largest knowledge base. This work represents the first practical constraint-based type inference technique for real-world code snippets, offering improvements in both scalability and performance.

Chapter 6

Conclusions and Future Work

In this chapter, we summarize the key findings of this thesis and discuss directions for future research.

6.1 Conclusion

This thesis presents three studies aimed at improving code snippet reusability: the development of SnR and Scitix for automatically inferring types in Java code snippets, and the creation of the ThaliaType benchmark suite for evaluating LLMs on type inference. By leveraging constraint-based type inference, SnR and Scitix enable precise, efficient, and explainable type inference for code snippets. Notably, Scitix demonstrates strong type inference performance while utilizing a large knowledge base, outperforming LLMs in both precision and recall.

In §3, we demonstrated that constraints extracted from code snippets can effectively guide a Datalog constraint solver for type inference. SnR precisely inferred missing types and successfully recovered import statements in the StatType-SO benchmark suite, improving both the compilability and consequently the reusability of these code snippets.

In §4, we evaluated LLMs' capabilities for type inference. Given the public availability of StatType-SO since 2017, there is a significant risk of data leakage into LLM training data. We confirmed data leakage in the open-source model StarCoder2, and introduced ThaliaType, along with transformations, to assess LLMs' generalization capabilities on unseen code snippets. All evaluated LLMs showed performance degradation on ThaliaType, consistent with the degradation observed in StarCoder2. More concerning was the performance decline seen when applying all transformations to StatType-SO code snippets. This decline, not observed on ThaliaType, aligns with the expected effects of data leakage on StatType-SO. Future evaluations of LLM-based type inference should account for the risk of data leakage and employ benchmark suites like ThaliaType that contain unseen code.

In §5, we addressed the scalability challenge in type inference, particularly in the presence of unknown types. By introducing the any type and iteratively adding constraints unrelated to unknown types, Scitix effectively handled real-world code snippets from StatType-SO and ThaliaType while scaling to a large knowledge base with over 3,000 jar files. Moreover, Scitix outperformed LLMs on StatType-SO, despite possible data leakage, and greatly outperformed LLMs on ThaliaType.

Collectively, this thesis contributes practical and effective solutions for enhancing code snippet reusability. It highlights critical challenges in evaluating LLMs for type inference, and proposes a precise, scalable constraint-based type inference technique to infer missing types, and to recover import statements, facilitating the reuse of incomplete code snippets.

6.2 Future Work

Several avenues can further advance type inference for Java code snippets.

First, while constraint-based type inference leverages the semantics of the code itself, it does not utilize auxiliary textual information such as code descriptions, accompanying comments, or metadata from online sources. For example, when developers reuse code snippets from platforms like Stack Overflow, the associated posts may reference libraries, frameworks, or contextual hints that could inform type inference, even if exact library versions remain unknown. Future work could explore parsing and integrating such textual cues to resolve ambiguities that code constraints alone cannot disambiguate.

Second, although each individual Datalog query used by Scitix is fast, inefficiencies arise when supertype constraints are added iteratively, as the Soufflé Datalog engine rechecks the entire constraint set during each iteration, including those previously validated. Optimizations such as caching previous satisfiability checks could reduce this overhead and further improve inference speed. While Scitix processes most code snippets within a few seconds using the largest knowledge base, certain snippets with a large number of supertype constraints take much longer due to the iterative process.

Third, advances in machine learning, such as LLMs, open promising avenues for type inference. LLMs, trained on vast amounts of real-world code, can capture common type usage patterns in snippets. LLMs performed well on popular types in unseen ThaliaType snippets. Combining the precision of constraint-based type inference with the learned knowledge embedded in LLMs may yield improved type inference performance.

Beyond type inference, further work is needed to support developers during code snippet reuse. This thesis primarily focused on recovering type information, but previous studies have shown that many Stack Overflow code snippets contain parsing errors that cannot be fixed by template-based repair, requiring manual intervention before type inference can proceed [53, 119]. Future research could investigate robust parsing techniques capable of automatically repairing or interpreting imperfect snippets, thereby reducing the manual effort required from developers during code snippet reuse.

References

[1] S. Subramanian, L. Inozemtseva, and R. Holmes, “Live api documentation,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–652. [Online]. Available: https://doi.org/10.1145/2568225.2568313

[2] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen, “Statistical learning of api fully qualified names in code snippets of online forums,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 632–642.

[3] C. M. K. Saifullah, M. Asaduzzaman, and C. K. Roy, “Learning from examples to find fully qualified names of api elements in code snippets,” in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’19. IEEE Press, 2020, p. 243–254. [Online]. Available: https://doi.org/10.1109/ASE.2019.00032

[4] Y. Dong, T. Gu, Y. Tian, and C. Sun, “Snr: Constraint-based type inference for incomplete java code snippets,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1982–1993. [Online]. Available: https://doi.org/10.1145/3510003.3510061

[5] Q. Huang, Z. Yuan, Z. Xing, X. Peng, X. Xu, and Q. Lu, “Fqn inference in partial code by prompt-tuned language model of code,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, dec 2023. [Online]. Available: https://doi.org/10.1145/3617174

[6] C. Velázquez-Rodríguez, D. Di Nucci, and C. De Roover, “A text classification approach to api type resolution for incomplete code snippets,” *Sci. Comput. Program.*, vol. 227, no. C, apr 2023. [Online]. Available: https://doi.org/10.1016/j.scico.2023.102941

[7] A. Kabir, S. Wang, Y. Tian, T.-H. P. Chen, M. Asaduzzaman, and W. Zhang, "Zs4c: Zero-shot synthesis of compilable code for incomplete code snippets using llms," *ACM Trans. Softw. Eng. Methodol.*, Nov. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3702979

[8] Z. Chen, A. Li, N. Zhang, J. Chen, Y. Huang, and Z. Zheng, "ijtyper: An iterative type inference framework for java by integrating constraint- and statistically-based methods," 2024. [Online]. Available: https://arxiv.org/abs/2402.09995

[9] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[10] M. Stonebraker, *Readings in database systems.* Morgan Kaufmann Publishers Inc., 1988.

[11] O. De Moor, G. Gottlob, T. Furche, and A. Sellers, *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers.* Springer, 2012, vol. 6702.

[12] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: An interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1213–1216. [Online]. Available: https://doi.org/10.1145/1989323.1989456

[13] N. Allen, P. Krishnan, and B. Scholz, "Combining type-analysis with points-to analysis for analyzing java library source-code," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–18. [Online]. Available: https://doi.org/10.1145/2771284.2771287

[14] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 243–262. [Online]. Available: https://doi.org/10.1145/1640089.1640108

[15] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA:

Association for Computing Machinery, 2006, p. 308–319. [Online]. Available: https://doi.org/10.1145/1133981.1134018

[16] S. Dawson, C. R. Ramakrishnan, and D. S. Warren, "Practical program analysis using general purpose logic programming systems—a case study," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 117–126. [Online]. Available: https://doi.org/10.1145/231379.231399

[17] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," vol. 28, no. 2, Feb. 2019. [Online]. Available: https://doi.org/10.1145/3295739

[18] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 131–144. [Online]. Available: https://doi.org/10.1145/996841.996859

[19] K. Hoder, N. Bjørner, and L. De Moura, "µz: an efficient engine for fixed points with constraints," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 457–462.

[20] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 422–430.

[21] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1371–1382. [Online]. Available: https://doi.org/10.1145/2723372.2742796

[22] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1135–1149. [Online]. Available: https://doi.org/10.1145/2882903.2915229

[23] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, “Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses,” in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 25–30. [Online]. Available: https://doi.org/10.1145/3088515.3088522

[24] OpenAI, “Gpt-4 technical report,” 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[25] Meta, “The llama 3 herd of models,” 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[26] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, “Starcoder 2 and the stack v2: The next generation,” 2024. [Online]. Available: https://arxiv.org/abs/2402.19173

[27] Y. Wei, F. Cassano, J. Liu, Y. Ding, N. Jain, Z. Mueller, H. de Vries, L. von Werra, A. Guha, and L. Zhang, “Selfcodealign: Self-alignment for code generation,” in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 62 787–62 874. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/72da102da91a8042a0b2aa968429a9f9-Paper-Conference.pdf

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[29] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, “Circle: continual repair across programming languages,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser.

ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 678–690. [Online]. Available: https://doi.org/10.1145/3533767.3534219

[30] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 535–547. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00063

[31] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[32] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 819–831. [Online]. Available: https://doi.org/10.1145/3650212.3680323

[33] Y. Ouyang, J. Yang, and L. Zhang, "Benchmarking automated program repair: An extensive study on both real-world and artificial bugs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 440–452. [Online]. Available: https://doi.org/10.1145/3650212.3652140

[34] I. Bouzenia, P. Devanbu, and M. Pradel, " RepairAgent: An Autonomous, LLM-Based Agent for Program Repair ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 2188–2200. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00157

[35] H. Ye, A. Z. Yang, C. Hu, Y. Wang, T. Zhang, and C. Le Goues, "Adverintent-agent: Adversarial reasoning for repair based on inferred program intent," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3728939

[36] J. Kong, X. Xie, M. Cheng, S. Liu, X. Du, and Q. Guo, "Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs,"

ACM Trans. Softw. Eng. Methodol., Mar. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3719345

[37] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3672456

[38] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660810

[39] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3690635

[40] L. Fan, Z. Liu, H. Wang, L. Bao, X. Xia, and S. Li, "Fait: Fault-aware fine-tuning for better code generation," 2025. [Online]. Available: https://arxiv.org/abs/2503.16913

[41] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, " Refactoring Programs Using Large Language Models with Few-Shot Examples ," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2023, pp. 151–160. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/APSEC60848.2023.00025

[42] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig, "Next-generation refactoring: Combining llm insights and ide capabilities for extract method," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 275–287.

[43] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: a long-range pre-trained language model for code completion," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[44] D. Wu, W. U. Ahmad, D. Zhang, M. K. Ramanathan, and X. Ma, "Repoformer: selective retrieval for repository-level code completion," ser. ICML'24. JMLR.org, 2024.

[45] A. Semenkin, V. Bibaev, Y. Sokolov, K. Krylov, A. Kalina, A. Khannanova, D. Savenkov, D. Rovdo, I. Davidenko, K. Karnaukhov, M. Vakhrushev,

M. Kostyukov, M. Podvitskii, P. Surkov, Y. Golubev, N. Povarov, and T. Bryksin, "Full line code completion: Bringing ai to desktop," 2025. [Online]. Available: https://arxiv.org/abs/2405.08704

[46] O. Sainz, J. A. Campos, I. García-Ferrero, J. Etxaniz, and E. Agirre, "Did chatgpt cheat on your test?" 2023. [Online]. Available: https://hitz-zentroa.github.io/lm-contamination/blog

[47] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[48] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[49] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[50] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 102–106. [Online]. Available: https://doi.org/10.1145/3639476.3639764

[51] J. Kong, X. Xie, and S. Liu, "Demystifying memorization in llm-based program repair via a general hypothesis testing framework," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729390

[52] A. Matton, T. Sherborne, D. Aumiller, E. Tommasone, M. Alizadeh, J. He, R. Ma, M. Voisin, E. Gilsenan-McMahon, and M. Gallé, “On leakage of code generation evaluation datasets,” 2024. [Online]. Available: https://arxiv.org/abs/2407.07565

[53] V. Terragni, Y. Liu, and S.-C. Cheung, “Csnippex: Automated synthesis of compilable code snippets from q&a sites,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 118–129. [Online]. Available: https://doi.org/10.1145/2931037.2931058

[54] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, “Do the dependency conflicts in my project matter?” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 319–330. [Online]. Available: https://doi.org/10.1145/3236024.3236056

[55] C. M. K. Saifullah, “Coster,” 2020. [Online]. Available: https://github.com/khaledkucse/COSTER

[56] C. M. K. Saifullah, M. Asaduzzaman, and C. Roy, “Coster: A tool for finding fully qualified names of api elements in online code snippets,” ser. ICSE ’21 DEMO, 03 2021.

[57] C. M. K. Saifullah, “Coster: A tool for finding fully qualified names of api elements in online code snippets,” 2020. [Online]. Available: https://youtu.be/oDZtw9MzUWM?t=208

[58] cianBuckley, “java - joda time converting time zoned date time to milliseconds - stack overflow,” 2013. [Online]. Available: https://web.archive.org/web/20170227042935/http://stackoverflow.com/questions/18274902/jodatime-converting-time-zoned-date-time-to-millis

[59] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, “Deepdelta: Learning to repair compilation errors,” 2019.

[60] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, “What makes a good code example?: A study of programming q a in stackoverflow,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34.

[61] E. Wong, Jinqiu Yang, and Lin Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 562–567.

[62] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 102–111.

[63] S. Baltes and S. Diehl, "Usage and attribution of stack overflow code snippets in github projects," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1259–1295, 2019.

[64] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and supporting adaptation of online code examples," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 316–327.

[65] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," in *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 146–161. [Online]. Available: https://doi.org/10.1145/117954.117965

[66] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach, "Making type inference practical," in *ECOOP '92 European Conference on Object-Oriented Programming*, O. L. Madsen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 329–349.

[67] T. Wang and S. F. Smith, "Precise constraint-based type inference for java," in *ECOOP 2001 — Object-Oriented Programming*, J. L. Knudsen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 99–117.

[68] D. Greenfieldboyce and J. S. Foster, "Type qualifier inference for java," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 321–336. [Online]. Available: https://doi.org/10.1145/1297027.1297051

[69] D. Smith and R. Cartwright, "Java type inference is broken: Can we fix it?" in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 505–524. [Online]. Available: https://doi.org/10.1145/1449764.1449804

[70] A. Aiken and E. L. Wimmers, "Type inclusion constraints and type inference," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 31–41. [Online]. Available: https://doi.org/10.1145/165180.165188

[71] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–5. [Online]. Available: https://doi.org/10.1145/3196398.3196450

[72] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 152–162. [Online]. Available: https://doi.org/10.1145/3236024.3236051

[73] R. S. Malik, J. Patra, and M. Pradel, "Nl2type: Inferring javascript function types from natural language information," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 304–315.

[74] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 47–57.

[75] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: Any snippets there?" in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 280–290.

[76] S. S. Manes and O. Baysal, "How often and what stackoverflow posts do developers reference in their github projects?" in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 235–239.

[77] A. W. Wong, A. Salimi, S. Chowdhury, and A. Hindle, "Syntax and stack overflow: A methodology for extracting a corpus of syntax errors and fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 318–322.

[78] V. Terragni and P. Salza, “Apization: Generating reusable apis from stackoverflow code snippets,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 542–554.

[79] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[80] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[81] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3695988

[82] H. A. Inan, O. Ramadan, L. Wutschitz, D. Jones, V. Rühle, J. Withers, and R. Sim, “Training data leakage analysis in language models,” February 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/training-data-leakage-analysis-in-language-models/

[83] “Github,” 2017. [Online]. Available: https://github.com/pdhung3012/TypeResolution_Oracle

[84] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, Ú. Erlingsson, A. Oprea, and C. Raffel, “Extracting training data from large language models,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2633–2650. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting

[85] M. Nasr, N. Carlini, J. Hayase, M. Jagielski, A. F. Cooper, D. Ippolito, C. A. Choquette-Choo, E. Wallace, F. Tramèr, and K. Lee, “Scalable extraction of training data from (production) language models,” 2023. [Online]. Available: https://arxiv.org/abs/2311.17035

[86] I. Ozkaya, “Application of large language models to software engineering tasks: Opportunities, risks, and implications,” *IEEE Software*, vol. 40, no. 3, pp. 4–8, 2023.

[87] “Models - openai api,” 2024. [Online]. Available: https://platform.openai.com/docs/models

[88] “Llm prompting guide,” 2024. [Online]. Available: https://huggingface.co/docs/transformers/en/tasks/prompting

[89] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, “Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556912

[90] T. Sotiropoulos, S. Chaliasos, and Z. Su, “Api-driven program synthesis for testing static typing implementations,” *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3632904

[91] “Overview - openai api,” 2024. [Online]. Available: https://platform.openai.com/docs/overview

[92] “Ollama,” 2024. [Online]. Available: https://ollama.com/

[93] D. Huang, Q. Bu, J. Zhang, X. Xie, J. Chen, and H. Cui, “Bias testing and mitigation in llm-based code generation,” 2024. [Online]. Available: https://arxiv.org/abs/2309.14345

[94] Y. Liu, X. Chen, Y. Gao, Z. Su, F. Zhang, D. Zan, J.-G. Lou, P.-Y. Chen, and T.-Y. Ho, “Uncovering and quantifying social biases in code generation,” in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS ’23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[95] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE’13, 2013, pp. 422–431.

[96] R. Dyer, H. Rajan, and T. N. Nguyen, “Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes,” in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, ser. GPCE, 2013, pp. 23–32.

[97] F. Wilcoxon, *Individual Comparisons by Ranking Methods.* New York, NY: Springer New York, 1992, pp. 196–202. [Online]. Available: https://doi.org/10.1007/978-1-4612-4380-9_16

[98] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java® Language Specification, Java SE 21 Edition*, Oracle, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/jls-3.html#jls-3.9

[99] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[100] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: Neural type prediction with search-based validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 209–220.

[101] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu, "Static inference meets deep learning: a hybrid type inference approach for python," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2019–2030.

[102] O. Sainz, J. Campos, I. García-Ferrero, J. Etxaniz, O. L. de Lacalle, and E. Agirre, "NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 10 776–10 787. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.722

[103] S. Golchin and M. Surdeanu, "Time travel in llms: Tracing data contamination in large language models," *arXiv preprint arXiv:2308.08493*, 2023.

[104] I. Magar and R. Schwartz, "Data contamination: From memorization to exploitation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 157–165. [Online]. Available: https://aclanthology.org/2022.acl-short.18

[105] S. Balloccu, P. Schmidtová, M. Lango, and O. Dušek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2024.

[106] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar, "Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models," 2024. [Online]. Available: https://arxiv.org/abs/2410.05229

[107] J. Cao, Z. Chen, J. Wu, S.-C. Cheung, and C. Xu, "Javabench: A benchmark of object-oriented code generation for evaluating large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 870–882.

[108] M. N. Uddin, A. Saeidi, D. Handa, A. Seth, T. C. Son, E. Blanco, S. R. Corman, and C. Baral, "Unseentimeqa: Time-sensitive question-answering beyond llms' memorization," 2025. [Online]. Available: https://arxiv.org/abs/2407.03525

[109] A. Elangovan, J. He, and K. Verspoor, "Memorization vs. generalization : Quantifying data leakage in NLP performance evaluation," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, P. Merlo, J. Tiedemann, and R. Tsarfaty, Eds. Online: Association for Computational Linguistics, Apr. 2021, pp. 1325–1335. [Online]. Available: https://aclanthology.org/2021.eacl-main.113/

[110] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 31 210–31 227. [Online]. Available: https://proceedings.mlr.press/v202/shi23a.html

[111] B. Jiang, Y. Xie, Z. Hao, X. Wang, T. Mallick, W. J. Su, C. J. Taylor, and D. Roth, "A peek into token bias: Large language models are not yet genuine reasoners," *arXiv preprint arXiv:2406.11050*, 2024.

[112] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *Scheme and Functional Programming Workshop*, vol. 6, 2006, pp. 81–92.

[113] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming*. USA: Elsevier Science Inc., 2006.

[114] S. Arch, X. Hu, D. Zhao, P. Subotić, and B. Scholz, "Building a join optimizer for soufflé," in *Logic-Based Program Synthesis and Transformation: 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21–23, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 83–102. [Online]. Available: https://doi.org/10.1007/978-3-031-16767-6_5

[115] Y. Dong, Z. Xu, Y. Tian, and C. Sun, "Beyond memorization: Evaluating the true type inference capabilities of llms for java code snippets," 2025. [Online]. Available: https://arxiv.org/abs/2503.04076

[116] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *ECOOP 2014 – Object-Oriented Programming*, R. Jones, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281.

[117] "mypy - optional static typing for python," https://web.archive.org/web/20230118003508/https://mypy-lang.org/, 2023.

[118] A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "Identifying versions of libraries used in stack overflow code snippets," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 341–345.

[119] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: an analysis of stack overflow code snippets," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 391–402. [Online]. Available: https://doi.org/10.1145/2901739.2901767