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

YIWEN DONG, University of Waterloo, Canada
ZHENYANG XU, University of Waterloo, Canada
YONGQIANG TIAN*, Monash University, Australia
EDWARD LEE, University of Toronto at Scarborough, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada
CHENGNIAN SUN, University of Waterloo, Canada

Code snippets commonly appear in online developer communities, documentation, and LLM-assisted workflows to communicate ideas and algorithms. However, *contextual* information, like *dependencies* and the *exact types*, are often missing in code snippets, which makes their reuse difficult. Some of the most successful automated techniques use logical constraints to infer the types and dependencies, but they do not work in practice because they require an exact knowledge base that contains *all* possible dependencies and exact types. However, such a knowledge base is both *computationally expensive* for constraint solving and *impossible to achieve* in the presence of *missing types* (*e.g.*, user-defined types) in code snippets.

To this end, this paper proposes a novel, scalable technique named Scitix. Our insight is two-fold. First, inspired by gradual typing's use of an unknown type, we represent certain missing types as `Any`, ignoring such types during constraint solving, improving performance and scalability. Second, our novel, iterative constraint-solving approach saves on computation and skips constraints involving missing types. Our extensive evaluations show that our insights improve both performance and scalability compared to SnR (the state of the art). Specifically, Scitix achieves F1-scores of 94.8% and 86.8% on Stack Overflow and generated code snippets, respectively, using a large knowledge base of over 3,000 jars. In contrast, SnR consistently times out, yielding near 0% F1. Even with the smallest knowledge base, where SnR does not time out, Scitix reduces the number of errors by 77% and 45% compared to SnR. Compared to state-of-the-art large language models (LLMs) like GPT-4o and the LLM-based ZS4C, Scitix improves F1-score by 76.8% and 35.4%, respectively. Scitix's strong performance highlights its potential as a practical technique for type inference in real-world code snippets.

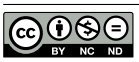
CCS Concepts: • **Software and its engineering** → *Automated static analysis*; • **Theory of computation** → *Type structures.*

Additional Key Words and Phrases: incomplete code snippets, type inference, constraint satisfaction, datalog

ACM Reference Format:
Yiwen Dong, Zhenyang Xu, Yongqiang Tian, Edward Lee, Ondřej Lhoták, and Chengnian Sun. 2026. Scitix: Scalable Constraint-Based Type Inference for Code Snippets with Missing Types. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA033 (October 2026), 24 pages. https://doi.org/10.1145/3832124

*Corresponding author

Authors' Contact Information: Yiwen Dong, University of Waterloo, Waterloo, Canada, yiwen.dong@uwaterloo.ca; Zhenyang Xu, University of Waterloo, Waterloo, Canada, zhenyang.xu@uwaterloo.ca; Yongqiang Tian, Monash University, Melbourne, Australia, yongqiang.tian@monash.edu; Edward Lee, University of Toronto at Scarborough, Scarborough, Canada, edwardi.lee@utoronto.ca; Ondřej Lhoták, University of Waterloo, Waterloo, Canada, olhotak@uwaterloo.ca; Chengnian Sun, University of Waterloo, Waterloo, Canada, cnsun@uwaterloo.ca.



ACM 2994-970X/2026/10-ARTISSTA033
https://doi.org/10.1145/3832124

```
1  public void to_reminder(View view) {
2      Intent intent = new Intent(this, Notification_morning.class);
3      AlarmManager manager = (AlarmManager) getSystemService(Activity.ALARM_SERVICE);
4      PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);
5      Calendar cal = Calendar.getInstance();
6      cal.set(Calendar.HOUR_OF_DAY, timepicker.getCurrentHour());
7      cal.set(Calendar.MINUTE, timepicker.getCurrentMinute());
8      cal.set(Calendar.SECOND, 0);
9      cal.set(Calendar.MILLISECOND, 0);
10     manager.setRepeating(AlarmManager.RTC_WAKEUP, cal.getTimeInMillis(),
11                          24*60*60*1000, pendingIntent);
12 }
```

Fig. 1. Formatted Stack Overflow code snippet #14241439 showing an event handler method called by Android with the relevant but unused `View` object.

1 Introduction

Code snippets are fragments of code commonly seen on developer forums. They are often used by software developers to illustrate ideas and algorithms that can be reused by other developers in other programs. But consider, how can a developer reuse the code snippet in Fig. 1?

The answer is less obvious than it seems. For one, a developer needs to *understand* what this fragment does. At first glance, it appears that this code snippet is creating a reminder widget that will be displayed on some `View` on a schedule set by some `Calendar`. But which concrete types do `View` and `Calendar` refer to? This depends on the larger source file from which this fragment is taken, which is missing in the fragment. This problem is not limited to code copied from developer forums. Modern development workflows also produce incomplete snippets through LLM-generated code, documentation examples, and issue discussions, where exact imports and dependencies are often absent. Before these fragments can be compiled, checked by an IDE, or integrated into a project, their referenced types must be resolved to concrete types.

To address this issue, several techniques have been proposed to infer types, uniquely identified by their fully qualified names (FQNs), automatically from code snippets [7, 12, 23, 26, 30, 33]. Among these approaches, constraint-based type inference [7, 30] offers a precise and explainable approach for inferring types from code snippets. It leverages the syntactic information present in the original code snippet to extract constraints on what the types can be. For example, the type `timepicker` shown in Fig. 1 should have a method named `getCurrentHour`, which takes no argument. To solve all the extracted constraints, a constraint-based type inference tool requires knowledge of existing types. For example, SnR is the state-of-the-art constraint-based type inference tool for incomplete Java code snippets. It solves the constraints extracted from the code snippets against a large knowledge base of FQNs collected from a number of commonly used libraries such as the Java Standard Library [20] and the Android SDK [2]. SnR infers in totality the set of solutions that satisfy all the constraints extracted from the code snippet, if such a solution exists. 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 [32].

While SnR is efficient when the size of the knowledge base is small (the evaluation in the original paper uses a knowledge base with 49 jars), its efficiency drastically decreases when the knowledge base expands. This limitation stems from SnR improperly handling the *missing types*, *i.e.*, types in the code snippet that no type in the knowledge base can fit in. When there are missing 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 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 instance, SnR takes 49 minutes to enumerate possible solutions for Fig. 1 when 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 missing 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 [29], we borrow the intuition behind gradual typing's unknown type and propose a special `Any` type for representing missing types within code snippets. `Any` lets Scitix avoid over-constraining inference, providing 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 *missing* from 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 missing types that cannot be satisfied.

When given a code snippet, Scitix automatically recommends FQNs for the types used. These recommendations can then be used directly by developers or passed to machine learning-based refinement steps such as LLMs-based repair, or used as part of a hybrid approach such as iJTyper [6].

We extensively evaluated Scitix using knowledge bases of varying sizes, comparing its performance against state-of-the-art techniques, including the constraint-based SnR, LLMs, and the LLM-based approach ZS4C. Our evaluation employed two established benchmarks. The first, ThaliaType [9], is a benchmark suite specifically designed for evaluating LLM-based type inference approaches while mitigating data leakage. The second, StatType-SO [23], is a widely used [6, 7, 12, 15, 23, 26, 33] benchmark suite that consists of manually repaired Stack Overflow code snippets with corresponding ground-truth types. However, because these snippets along with their ground truth types have been exposed to LLMs during training [9], StatType-SO suffers from *data leakage*, which falsely inflates LLM performance and renders StatType-SO unsuitable for evaluating LLM-based approaches. Since Scitix does not rely on LLMs, StatType-SO provides a reliable measure of its real-world effectiveness. Scitix outperformed all baselines, with a 35.4% higher F1-score than ZS4C and a 76.8% higher F1-score than GPT-4o. The comparison with SnR shows that Scitix scales better as the knowledge base grows. 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 achieved F1-scores of 86.8% on ThaliaType and 94.8% on StatType-SO. Even with the smallest knowledge base, Scitix improved F1-score over SnR by 7.7% on ThaliaType and 4.7% on StatType-SO.

Contribution. This paper makes the following contributions.

- We propose Scitix, a novel, scalable approach for constraint-based type inference that efficiently and effectively handles missing types in code snippets via the `Any` type and an iterative approach.
- We demonstrate that Scitix outperforms state-of-the-art LLM-based approaches on ThaliaType, with a 35.4% higher F1-score than ZS4C and a 76.8% higher F1-score than GPT-4o.
- We show that Scitix scales better than SnR, achieving F1-scores of 86.8% on ThaliaType and 94.8% on StatType-SO with 3,000+ jars in the knowledge base, while SnR's scores fell near zero.
- To facilitate reproducibility and replicability, we provided a replication package [**?**].

2 Motivating Example

We use Fig. 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.

Scitix and prior work [7, 13, 23, 26, 30, 33] 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 missing types such as `timepicker` and `Notification_morning`, which complicate step (a).

To infer the types in Fig. 1 using constraint-based type inference, SnR follows 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 Fig. 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.

- `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 1 shows results from querying the knowledge base using the simple names in Fig. 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 queried type is missing (*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, or if the correct type is missing from the knowledge base.

For `Calendar`, the query resulted in scenario ②, yielding multiple `Calendar` types across libraries. SnR then uses constraint solving to narrow this set of types down to a single satisfying type.

Solve Constraints. Below is a simplified list of facts used when solving constraints. To improve clarity, less relevant classes and 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` returning `Calendar` and `long` types, respectively. Additionally, the `sundial` library contains a class with the FQN `org.quartz.core.Calendar`, and the `Calendar` simple name, but without the relevant methods.

Table 1. Query result for FQNs using the simple names in Fig. 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

Simple Name	FQN
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("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 $\tau_1 \longrightarrow$ `"jdk:java.util.Calendar"`. Using this mapping, SnR can correctly add `import java.util.Calendar`.

However, the presence of missing types makes the entire set of constraints unsatisfiable. While the type under scenario ① can be reasonably considered missing, finding exactly which type is missing under scenario ② is impossible without the ground truth. The way SnR addresses the issue is to guess which type is missing 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 missing type correctly, thus further losing the precision and recall.

2.1 Limitations of SnR

SnR models type inference as a Constraint Satisfaction Problem (CSP) [25]. 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 missing types, which leads to reduced precision in inference. We illustrate this issue using line 2 in Fig. 1. To start, the line is annotated with TVs representing the used types.

`new Intent(this, Notification_morning.class);` → `new` τ_1`(`τ_2`, `τ_3`);`

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; assuming 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(`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`)`

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

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

τ_1	→	`"android:android.content.Intent"`	τ_4	→	`"jdk:void"`
τ_2	→	??	τ_5	→	`"android:android.content.Context"`
τ_3	→	`"jdk:java.lang.Class"`	τ_6	→	`"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 Fig. 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 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.

2.2 A Glance at Scitix

Although the constraints extracted from Fig. 1 are unsatisfiable as a whole, 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 constraints, and focus on the remaining constraints for `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`)`

Now the remaining constraints are satisfiable with the original facts above in §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 generate facts for every type τ_5 can potentially represent, this is impractical. The TV τ_5 can 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 deletes all `supertype` constraints and refines the solution by adding back `supertype` constraints iteratively. As long as adding the `supertype` 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 constraints, that `Intent` must be from `Android`.

3 Methodology

Fig. 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.

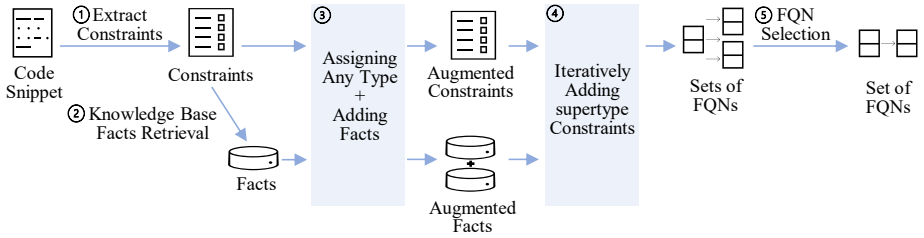


Fig. 2. The workflow of Scitix.

Table 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.

In step ①, constraints are extracted from the code snippet (§3.1). In step ②, relevant facts are retrieved from the knowledge base (§3.2). During step ③, Scitix assigns `Any` type to the types in the code snippet that can be identified as missing, and adds the required facts after the assignment (§3.3). To efficiently handle the missing 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, §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, §3.6) and used to add missing import statements and libraries to the code snippet.

3.1 Extracting Constraints

Table 2 shows the constraints used by Scitix. To extract the initial constraints from the AST, Scitix employs SnR, which performs static analysis based on the rules outlined in Table 3. The table summarizes representative rules illustrating Scitix's constraint solving, rather than all rules inherited from SnR. To improve the quality of the extracted constraints, SnR leverages the knowledge base to resolve ambiguities in the AST. For example, in `Calendar.getInstance()`, to differentiate if `Calendar` is a type or an undeclared variable and if a class constraint should be emitted, SnR queries the knowledge base. If the knowledge base contains a type with the simple name `Calendar`, then SnR assumes `Calendar` is a type. Scitix adopts the same heuristic as SnR, but improves robustness when it fails. If the heuristic is incorrect (*e.g.*, `Calendar` is an undeclared variable) and the resulting constraints become unsatisfiable, Scitix assigns `Calendar` the `Any` type to proceed.

3.2 Retrieving Facts from Knowledge Base

Next, all the existing known facts for FQNs that the types in the code snippet might represent are extracted from the knowledge base and given to Scitix. Table 4 details the various kinds of facts stored in the knowledge base.

Parameterized Types. In SnR, the facts retrieved from the knowledge base are also augmented using gathered constraints. This process adds parameterized types and their corresponding methods, which are then passed to Scitix. For example, `Notification_morning.class` is a parameterized type `Class<Notification_morning>`. Initially, the knowledge base contains these facts for `Class<T>`.

Table 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
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")

- `class("jdk:java.lang.Class", "java.lang.Class", "Class")`
- `method(0, "jdk:java.lang.Class", "T", "newInstance")`

To create a `Class<Notification_morning>` from `Class<T>`, the type parameters are replaced by the instantiating type (`T` replaced by `Notification_morning`). This process yields these new facts.

- `class("jdk:java.lang.Class<Notification_morning>", "java.lang.Class<Notification_morning>", "Class<Notification_morning>")`
- `method(0,"jdk:java.lang.Class<Notification_morning>", "Notification_morning", "newInstance")`

In the type `Class<Notification_morning>`, there is now a `newInstance` method that returns a `Notification_morning`. This process is repeated for all the parameterized types that appear directly in the code snippet, such as those in variable declarations. These parameterized types are added to the knowledge base and the augmented facts are handed to Scitix. As a result, Scitix is able to infer parameterized types as well.

3.3 Assigning the `Any` Type and Adding Facts

As mentioned in §2, there may be no type in the knowledge base that corresponds to the type represented in the code snippet. That is, 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 a missing type. To handle these identified missing 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 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, the generated facts state that the `Any` type has the same simple name and the same method. To allow the `Any` type and the return type of the method to be

Table 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 are omitted to ease presentation.

Fact Type	Attributes	Description
`class`	(Type, FQN, sn)	Type has the given FQN and simple name (sn).
`supertype`	(Type_1, Type_2)	Type_1 is super type of Type_2.
`method`	(id, Type, 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. Augmented constraints and generated facts for assigning type variables to the Any type. Remaining variables (e.g., id, FQN, 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)

resolved independently, Scitix also augments the method constraint to decouple these two types. The augmented method constraint is also shown in Table 5.

This process of assigning the Any type targets constraints that are directly derived from the simple names and method calls used in the code snippet, such as class, method, and method_param. Compared to supertype constraints, these constraints can be queried directly from the knowledge base, making them more efficient to solve. When no solution is found in the knowledge base for a type variable, it indicates that the type is missing. Scitix leverages this insight to identify some of the missing types in the snippet and assigns them the Any type for more efficient constraint solving.

3.4 Iteratively Adding supertype Constraints

Although some missing types can be identified and handled in the last step, there can still be some missing 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. 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, which can be verified using a Datalog solver, then the constraint 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.

This approach contrasts with the method described in §3.3 where Any types facts were added to handle the missing 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 missing 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 iteratively adding the `supertype` constraints, the unsatisfiable `supertype` constraints that arise from missing types and cannot be precisely identified in §3.3 are now decoupled from the constraint-solving process.

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 [14], 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 [3]. 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 Fig. 1). Moreover, method calls can return types without explicitly mentioning a simple name, even from other libraries (transient dependency). Since SnR cannot handle missing types and unsatisfiable constraints well, it considers all libraries using both simple names and method names, at the cost of scalability.

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 [7]. 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. Following CSNIPPEX's clustering hypothesis [31], which observes that referred classes often come from the same libraries, the library score favors mappings whose resolved types are concentrated within the selected libraries.

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 Fig. 1, `View`, with only a `class` constraint, matched multiple types in the knowledge base in Table 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.

4 Evaluations

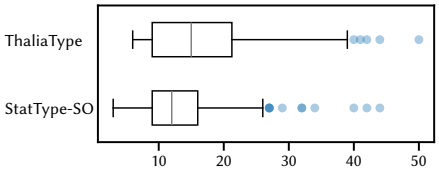
Our evaluation was guided by the following research questions:

- RQ4 How does Scitix compare to the state-of-the-art methods for import inference?
- 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 impact of different components of Scitix on its overall performance and runtime?
- RQ5 How well does Scitix perform in the presence of missing types?

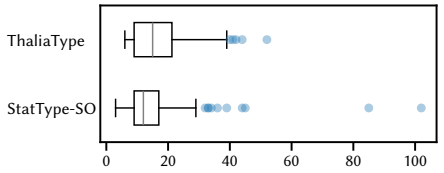
RQ1 provides a holistic comparison against all baselines, while RQs 2–5 provide a deeper analysis of scalability, ablation, and the impact of missing types.

4.1 Experiment Setup

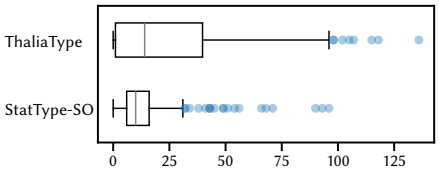
Scitix. We implemented Scitix in Java. In Scitix, MariaDB served as the knowledge base; initial constraints were provided by SnR; and constraints were solved by the Soufflé Datalog solver. A replication package is available at https://doi.org/10.5281/zenodo.21018232.



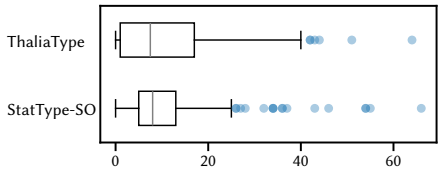
(a) Comparing the number of type variables.



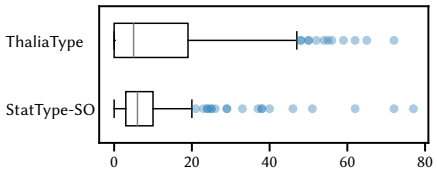
(b) Comparing the number of class constraints.



(c) Comparing the number of super type constraints.



(d) Comparing the number of method constraints.



(e) Comparing the number of method parameter constraints.

Fig. 3. Boxplots comparing the number of type variables and constraints in ThaliaType versus StatType-SO.

4.1.1 Baseline Approaches. We evaluated Scitix against state-of-the-art LLM-based and constraint-based baselines.

ZS4C. We compared against ZS4C [15], a state-of-the-art baseline that uses an LLM with compiler feedback to infer import statements in code snippets. ZS4C can be tuned for the number of self-consistency samples (K) and the number of LLM compilation-fix iterations (M). We used the best observed setting ($K = 10, M = 20$) with GPT-4o as the underlying LLM. **??** reports the parameter sweeps used to select this ZS4C setting for comparison.

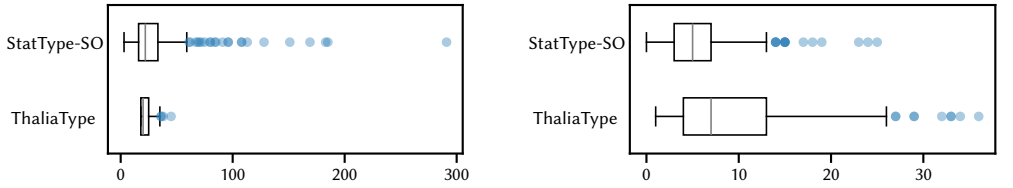
Plain LLM-based approaches. We also benchmarked against state-of-the-art LLM models including the closed-source GPT-4o and GPT-4o-mini [19], open-weight models Llama3.1:70b and Llama3.1:8b [17], and the open-source model Starcoder2:15b [**? ?**].

Constraint-based. We also compared against the state-of-the-art constraint-based technique SnR using its publicly available replication package [8].

4.1.2 Datasets and Knowledge Bases. We used two datasets and seven knowledge bases to evaluate Scitix's performance.

Dataset: ThaliaType. To ensure a fair evaluation against LLM-based approaches, we used *ThaliaType* [9], a dataset specifically designed to assess LLM type inference performance on code snippets. ThaliaType consists of 300 entirely new, previously unseen snippets.

Dataset: StatType-SO. Furthermore, we utilized *StatType-SO* [23], a widely adopted dataset for evaluating type inference tools for code snippets [7, 13, 23, 26, 33], including SnR. StatType-SO consists of 267 manually repaired Stack Overflow code snippets. Although StatType-SO is used to evaluate Scitix and SnR, it is unsuited for evaluating LLM-based approaches because the ground truths along with the code snippets are publicly available in multiple GitHub repositories since 2017. Starcoder2:15b, a model with open source training data, was directly observed to be trained



(a) Comparing lines of code. (b) Comparing the number of import statements.

Fig. 4. Boxplots comparing the code snippets in ThaliaType and StatType-SO.

Table 6. The number of code snippets using each language feature in ThaliaType and StatType-SO.

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

on all code snippets in StatType-SO [9]. We included StatType-SO to evaluate Scitix's performance on real-world code snippets.

Dataset Comparison. Both datasets utilize the same six libraries to ensure they are familiar to the LLMs. Looking at the code snippets in ThaliaType and StatType-SO, snippets from both datasets are comparable in length and number of import statements (Fig. 3). Both ThaliaType and StatType-SO exercise a diverse set of language features such as parameterized types and wildcard types (Table 6). While ThaliaType results in more type variables during type inference, it results in slightly fewer method and method parameter constraints compared to StatType-SO (Fig. 4).

Knowledge Bases. The initial knowledge base, denoted as Γ_0, was constructed using the 49 jars used by both ThaliaType and StatType-SO. Each knowledge base stores the facts described in Table 4. These facts are constructed from each jar by extracting each class's FQN, simple names, direct inheritance relations, method signatures, and method parameter types. To evaluate scalability, we progressively expanded the knowledge base by incorporating additional jar files. Following the procedure from prior work [32], 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}.

4.1.3 Configuration. 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. To perform the LLM-based experiments, Llama3.1:8b, Llama3.1:70b, and Starcoder2:15b models were hosted using an Ollama server backed by an RTX 6000 Ada GPU, while GPT-4o and GPT-4o-mini were accessed through the OpenAI APIs.

4.1.4 Metrics. To evaluate import inference performance, we computed precision, recall, and F1-scores following prior work [15] as follows.

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

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}`

Fig. 5. The prompt used to infer types on code snippets with LLMs. `{input_code}` is replaced by the snippet.

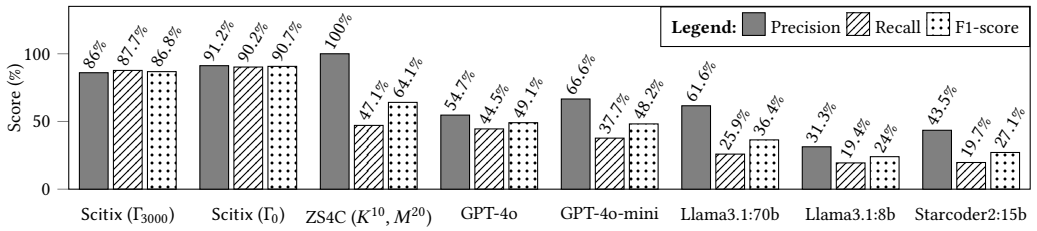


Fig. 6. Precision, recall, and F1-scores of Scitix on ThaliaType code snippets using Γ_{3000} and Γ_0 knowledge bases compared to ZS4C (with parameters $K = 10$ and $M = 20$), GPT-4o, GPT-4o-mini, Llama3.1:70b, Llama3.1:8b, and Starcoder2:15b.

4.2 RQ1: How does Scitix compare to state-of-the-art methods for import inference?

We evaluated Scitix's performance on ThaliaType code snippets without import statements. Because StatType-SO code snippets that were combined with ground truths have been publicly available on GitHub since 2017 and shown to be used for LLM training [9], StatType-SO cannot be used for comparison with LLMs.

To ensure a comprehensive and fair evaluation, we utilized a prompt from prior work (Fig. 8). Each code snippet was given to the LLMs without import statements, and the inferred import statements were collected.

Scitix outperformed the evaluated LLM-based baselines on ThaliaType (Fig. 9). Scitix reduced the recall error rate by 78% on ThaliaType compared to the best-performing plain LLM (GPT-4o). On unseen code snippets in ThaliaType, Scitix outperformed GPT-4o by 76.8% and ZS4C by 35.4% in F1-score. Notably, ZS4C achieved 100% precision, due in large part to compiler feedback that allowed incorrect import statements to be filtered out. This precision is higher than Scitix's, but ZS4C's lower recall led to a lower F1-score. Scitix does not use compiler feedback to remove incorrectly inferred types. Thus, a hybrid technique may benefit from both Scitix's scalability and ZS4C's high precision.

Against the state-of-the-art constraint-based baseline, Scitix also outperforms SnR across all knowledge base sizes (Fig. 6). RQ1 analyzes this constraint-based comparison in detail, and RQ2 examines the running time.

4.3 RQ2: How well does Scitix infer import statements as the knowledge base scales?

We next compare Scitix with SnR as the knowledge base grows, using both the ThaliaType and StatType-SO datasets. Across all knowledge base sizes, Scitix consistently outperformed SnR, particularly as the knowledge base size increased, demonstrating superior scalability. As shown in Figs. 5 and 6, Scitix reduced the percent of recall errors ($\frac{\text{incorrectly inferred}}{\text{all expected}}$) by 45% on ThaliaType and 77% on StatType-SO, relative to SnR, even on the smallest knowledge base (Γ_0). Moreover, Scitix achieved F1-scores of 90.7% (ThaliaType) and 97.6% (StatType-SO), compared to SnR's 84.2% and 93.2% on the original Γ_0 knowledge base.

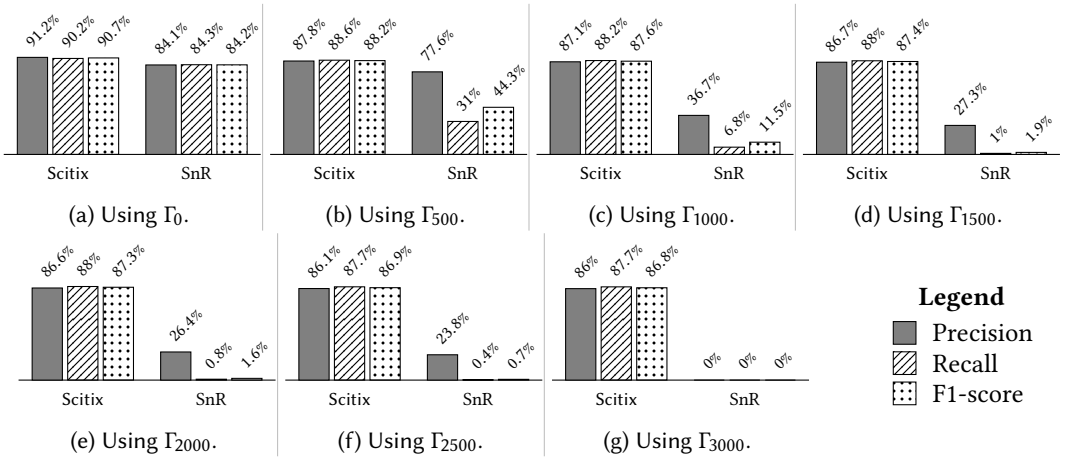


Fig. 7. Comparison of precision, recall, and F1-score on ThaliaType.

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 86.8% (ThaliaType) and 94.8% (StatType-SO), whereas SnR dropped to 0.0% and 2.4%, 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 missing 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 in the knowledge base, and 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.

Both SnR and Scitix perform worse on ThaliaType than on StatType-SO because there is less information such as methods to help narrow down the potential solutions in ThaliaType (Fig. 4). Simultaneously, there are more type variables to solve for. With fewer constraints per variable, selecting the correct FQN often becomes ambiguous.

4.4 RQ3: 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. Fig. 7 illustrates the time taken for Scitix and SnR to complete type inference on ThaliaType and StatType-SO 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 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. On Γ_0, Scitix took an average of 11 seconds (ThaliaType) and 10 seconds (StatType-SO). Scitix scaled efficiently with only a slight increase in runtime on Γ_{3000}, taking an average of 17 seconds for both ThaliaType and StatType-SO.

In contrast, while SnR performed well on Γ_0 (average of 9 seconds and 7 seconds on ThaliaType and StatType-SO respectively), SnR failed to scale. On Γ_{3000}, SnR averaged 274 seconds (ThaliaType) and 289 seconds (StatType-SO), failing to complete inference for 93% of the StatType-SO code

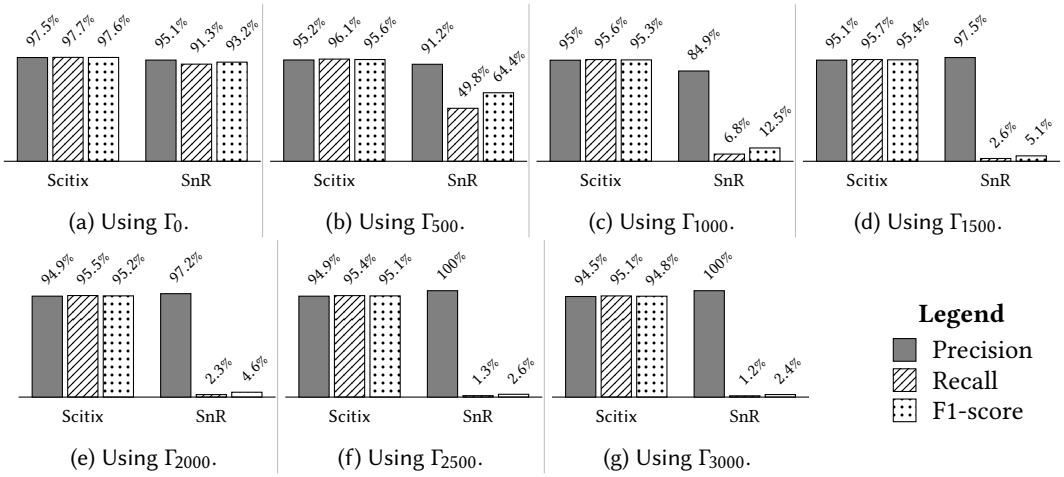
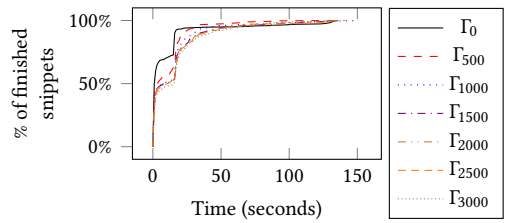
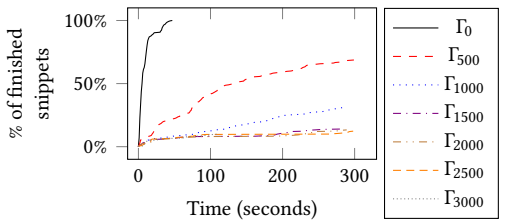


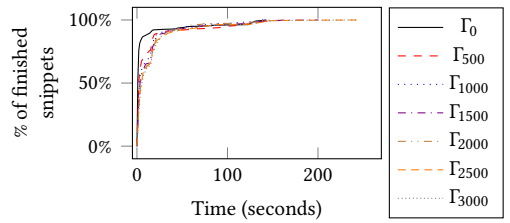
Fig. 8. Comparison of precision, recall, and F1-score on StatType-SO.



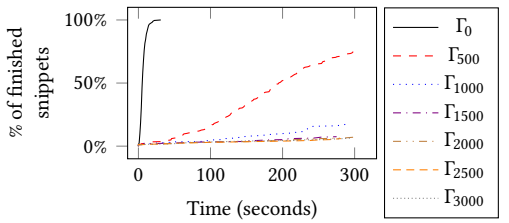
(a) Scitix on ThaliaType code snippets using various knowledge bases.



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



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



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

Fig. 9. 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.

snippets within the five-minute timeout. Notably, Scitix did not timeout on any code snippets in ThaliaType and StatType-SO.

4.5 RQ4: What is the impact of different components of Scitix on its overall performance and runtime?

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

Table 7. The precision, recall, and F1-scores completed by Scitix and its variants 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 bolded.

		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
ThaliaType	Scitix	91.2%	90.2%	90.7%	**87.8%**	**88.6%**	**88.2%**	**87.1%**	**88.2%**	**87.6%**	**86.7%**	88.0%	**87.3%**	**86.6%**	88.0%	**87.3%**	**86.1%**	**87.7%**	**86.9%**	**86.0%**	87.7%	**86.8%**
	Scitix $_{\text{Simple}}$	89.9%	89.0%	89.4%	86.3%	87.1%	86.7%	85.8%	86.9%	86.3%	85.4%	86.7%	86.1%	85.4%	86.8%	86.1%	84.8%	86.5%	85.6%	84.8%	86.4%	85.6%
	Scitix $_{\text{Random}}$	90.4%	90.0%	90.2%	87.5%	88.5%	88.0%	86.6%	**88.2%**	87.4%	86.4%	**88.1%**	87.2%	86.3%	**88.1%**	87.1%	85.9%	**87.8%**	86.8%	85.7%	**87.8%**	86.7%
	Scitix $_{\text{Filter}}$	**91.3%**	**90.3%**	**90.8%**	**87.8%**	**88.6%**	**88.2%**	87.0%	**88.2%**	**87.6%**	86.6%	88.0%	**87.3%**	86.5%	88.0%	87.2%	86.0%	87.7%	**86.9%**	85.9%	87.7%	**86.8%**
	Scitix $_{\text{Naive}}$	85.2%	84.6%	84.9%	76.3%	76.0%	76.1%	74.3%	74.6%	74.4%	71.2%	71.7%	71.5%	70.1%	70.6%	70.4%	69.0%	70.2%	69.6%	68.9%	69.7%	69.3%
StatType-SO	Scitix	**97.5%**	**97.7%**	**97.6%**	**95.2%**	**96.1%**	**95.6%**	**95.0%**	**95.6%**	**95.3%**	**95.1%**	**95.6%**	**95.4%**	**94.9%**	**95.4%**	**95.2%**	**94.9%**	**95.4%**	**95.1%**	94.5%	**95.1%**	**94.8%**
	Scitix $_{\text{Simple}}$	96.7%	96.8%	96.8%	94.6%	95.3%	94.9%	94.1%	94.6%	94.4%	93.8%	94.4%	94.1%	93.7%	94.3%	94.0%	93.7%	94.2%	94.0%	93.7%	94.3%	94.0%
	Scitix $_{\text{Random}}$	96.9%	**97.7%**	97.3%	94.4%	95.9%	95.1%	94.3%	95.5%	94.9%	94.4%	95.2%	94.8%	94.3%	95.2%	94.8%	94.4%	**95.4%**	94.9%	93.9%	**95.1%**	94.5%
	Scitix $_{\text{Filter}}$	97.2%	**97.7%**	97.4%	**95.2%**	**96.1%**	**95.6%**	94.9%	**95.6%**	**95.3%**	95.0%	94.4%	94.7%	**94.9%**	93.8%	94.3%	94.8%	93.6%	94.2%	**94.6%**	93.4%	94.0%
	Scitix $_{\text{Naive}}$	94.5%	94.6%	94.6%	86.8%	87.7%	87.2%	84.4%	85.2%	84.8%	82.8%	83.7%	83.3%	82.4%	83.2%	82.8%	82.7%	83.3%	83.0%	83.0%	83.9%	83.4%

Table 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.

		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	ThaliaType	10	14	13	15	15	17	17	StatType-SO	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

Table 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.

	ThaliaType Γ_0			ThaliaType Γ_R			StatType-SO Γ_0			StatType-SO Γ_R		
Tool	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Scitix	93.3%	**87.1%**	**90.1%**	90.0%	**85.7%**	**87.8%**	**97.3%**	**97.7%**	**97.5%**	**93.0%**	**96.3%**	**94.6%**
SnR	**93.5%**	83.0%	87.9%	**90.7%**	67.8%	77.6%	94.4%	91.5%	92.9%	91.2%	77.9%	84.0%

fixed, we created four variants. Scitix $_{\text{Simple}}$ omits super type constraints. Scitix $_{\text{Random}}$ randomizes the order of all generated constraints considered for each query (*i.e.*, `class`, `method`, `method_param`, and `supertype`). 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 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 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 Scitix with package filtering. Scitix $_{\text{Random}}$ performed the slowest while achieving nearly the same F1-scores as Scitix, suggesting that ordering mainly affects runtime rather than inference quality. Different orderings of `supertype` constraints can lead the greedy procedure to different locally maximal satisfiable sets, which explains the small quality differences.

4.6 RQ5: How well does Scitix perform in the presence of missing types?

Scitix is primarily designed for scalability (*i.e.*, incorporating significantly more types into the knowledge base), by accounting for missing types. This design, in turn, enhances its resilience to missing 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. Since in practice, when building a knowledge base, each type along with its dependencies are included together to ensure consistency. To avoid breaking these dependencies and to ensure the setup remained realistic, types referenced by another type were retained in Γ_R. Specifically, all types that were not referenced by any other type were removed. 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 9 shows that Scitix's F1-score dropped only slightly with additional missing types (2.6% on ThaliaType, 3.0% on StatType-SO) using the Γ_R knowledge base. On the other hand, SnR saw a much larger decrease (11.7% and 9.6%, respectively). Thus, Scitix is more resilient to incomplete knowledge bases, making it better suited for real-world scenarios.

5 Discussion

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 missing types from unsupported language features as `Any`. Our evaluation using StatType-SO and ThaliaType with 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.

Scitix also relies on library filtering to keep constraint solving scalable, but this filtering does not explicitly model library versions. Related partial-program work has used observed type, method, and field names to narrow candidate program or library versions [**?**]. While Scitix focuses on scalability over a large number of distinct libraries, future work could incorporate version-aware filtering to select more precise versions after the library family has been identified.

5.2 Soundness

Scitix does not guarantee soundness as given a code snippet and with all the necessary types in the knowledge base, Scitix may not infer the expected set of FQNs. The generated constraints approximate the relevant Java typing rules, but those rules are designed to check whether a completed program is well typed, not to uniquely infer imports from an incomplete code snippet. The result is sound only when the correct FQNs are present in the knowledge base and the generated constraints are sufficient to uniquely distinguish them from other compatible FQNs, such as candidates with the same simple names. The `Any` type is introduced only when the original constraints are already unsatisfiable, so it should be understood as a recovery mechanism without providing soundness guarantees. Our evaluation shows that this design substantially improves import inference for real-world code snippets despite the lack of a full soundness guarantee.

Table 10. ZS4C parameter sweeps. The best result for each column is bolded.

(a) Varying K with $M = 5$

K	P (%)	R (%)	F1 (%)
10	**99.8**	44.3	61.3
20	99.3	**45.7**	**62.6**
30	99.2	45.5	62.3

(b) Varying M with $K = 10$

M	P (%)	R (%)	F1 (%)
5	99.8	44.3	61.3
10	99.6	44.4	61.4
15	99.8	46.0	63.0
20	**100**	**47.1**	**64.1**

(c) Varying K with $M = 20$

K	P (%)	R (%)	F1 (%)
10	**100**	47.1	**64.1**
20	99.2	47.1	63.9
30	99.3	**47.2**	63.9

5.3 Threats to Validity

Internal. To mitigate potential threats to validity, we investigated cases where Scitix underperformed to ensure our implementation is 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 [**?**].

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 of methods and inheritance are widely used in object-oriented languages. Scitix can be easily adapted to support other languages by using a suitable constraint generator and knowledge base.

5.4 ZS4C Evaluation Setting

A fair comparison between Scitix and ZS4C requires accounting for ZS4C's tunable parameters, including the number of self-consistency samples (K) and compiler-feedback repair iterations (M). Since increasing M can improve recall by allowing more repair attempts, we examined multiple K and M settings before selecting the ZS4C configuration used in our evaluation.

?? shows three ZS4C parameter sweeps for the number of self-consistency samples (K) and the number of LLM compilation-fix iterations (M). The first two sweeps mirror the ZS4C paper's one-dimensional analysis. We vary K with $M = 5$ fixed (**??**) and vary M with $K = 10$ fixed (**??**). The third sweep varies K again, but with $M = 20$ fixed, because increasing M consistently improved F1. This final sweep checks whether additional self-consistency samples improve the best observed compilation-fix setting (**??**).

The original ZS4C paper evaluated up to $K = 30$ and $M = 15$. Because F1 continued to improve from $M = 10$ to $M = 15$, we extended the M sweep to $M = 20$. The gains diminished after $M = 15$, as increasing M from 15 to 20 produced only one additional successfully compiled snippet, while the potential number of LLM repair calls, and thus the API cost, grows linearly with M. We therefore stopped the M sweep at $M = 20$. At this setting, increasing K did not improve F1. Thus the best observed configuration was $K = 10, M = 20$, achieving 100% precision, 47.1% recall, and 64.1% F1.

6 Related Work

This section discusses type inference for Java code snippets, type inference for dynamic languages, partial program analysis, and gradual typing as a conceptual background for our use of `Any`.

Type Inference on Java Code Snippets. Type inference for Java code snippets has been extensively studied [6, 7, 12, 15, 23, 26, 30, 33**? ?**]. Baker [30] and SnR [7] leveraged constraint-based type inference to infer types on Java code snippets, following Java type system rules. Scitix builds on this line of work by incorporating the `Any` type, substantially improving precision, recall, and scalability on large, realistic knowledge bases.

Our work *complements* machine learning (ML)-based techniques [12, 23, 26, 33], and LLM-based techniques [6, 15**? ?**] which are commonly trained on GitHub project datasets. Scitix achieves higher performance at lower computational cost than ML-based techniques, including LLM-based techniques. Additionally, LLM-based techniques suffer from *data leakage*, a common issue in

engineering research [5, 9, 10, 18, 21, 27, 28, 34]. StatType-SO has been publicly available on GitHub since 2017 leading to data leakage [9], and making direct comparisons undependable.

Compared to ML-based techniques, Scitix is more explainable, with rules dictating the imported types. In hybrid approaches like iJTyper [6], which combine constraint-based type inference with ML-based methods, Scitix can strengthen the constraint-based component and refine ML outputs.

Reusing real-world Stack Overflow code snippets is challenging. Many studies have worked on large-scale repair and reuse [31, 32, 35]. CSNIPPEX [31] 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 [32] extended CSNIPPEX to create reusable APIs from Stack Overflow code. Zerouali et al. [35] 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.

Type Inference on Dynamically Typed Languages. Machine learning techniques have also been leveraged for type inference on dynamically typed languages [11, 16, 22, 24]. These techniques leverage additional sources of information such as comments, and naming conventions where as Scitix relied only on precise constraint-based type inference. Scitix targets statically typed languages and has shown to be highly precise. Future work may consider extending constraint-based type inference on dynamically typed languages.

Partial Program Analysis. PPA [**?**] and GRAPA [**?**] are rule-based approaches for incomplete Java programs. Like PPA and GRAPA, Scitix uses type-system-inspired rules, but its main novelty lies in scaling this rule-based reasoning to snippet inference over thousands of libraries. Unlike PPA-style work, which repairs incomplete programs to support downstream static analysis, Scitix focuses on recovering import statements and dependencies for code snippets over a large knowledge base while remaining robust to types that are absent from that knowledge base. Future work may explore further downstream analysis similar to prior PPA-based bug-signature inference [**?**] for code snippets.

Gradual Typing. Scitix draws on ideas from gradual typing, especially the use of an unknown-like type to represent missing type information during inference. Prior work on gradual typing and gradual type inference [29**? ? ? ? ?**] provides useful background on reasoning when some type information is unavailable. Scitix adapts this intuition to a different setting of constraint-based inference for incomplete Java snippets. Our use of `Any` is therefore only an inference-time mechanism, not a full gradual-typing treatment for Java. Instead, Scitix uses `Any` as a pragmatic placeholder for types in code snippets that are missing from the knowledge base, so that constraints involving those types do not prevent inference from continuing on the remaining code.

7 Conclusion

This study 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 missing types. Scitix marks types in a code snippet that are missing from the knowledge base as a special `Any` type and iteratively adds constraints to enhance precision. We conducted a comprehensive evaluation using ThaliaType to mitigate data leakage concerns, and StatType-SO to assess potential real-world performance. Scitix demonstrated great scalability, achieving F1-scores of 86.8% on ThaliaType and 94.8% on StatType-SO with the largest knowledge base while SnR consistently timed out. Furthermore, Scitix outperformed LLMs, improving F1 by 76.8% over GPT-4o and 35.4% over ZS4C 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.

Data-Availability Statement

The replication package for Scitix is publicly available at https://doi.org/10.5281/zenodo.21018232. It contains all evaluation scripts, binaries, libraries, and benchmarking results necessary to reproduce the findings reported in this paper. To facilitate replication, we also provide prebuilt Docker containers that include the replication package and all required dependencies, as described in the accompanying README file. Additionally, the package includes the knowledge bases and scraped jar files used in our experiments.

Acknowledgments

We thank the anonymous ISSTA 2026 reviewers for their constructive feedback and insightful comments. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, and by CFI-JELF Project #40736. Part of this research was supported by use of the Nectar Research Cloud, a collaborative Australian research platform supported by the NCRIS-funded Australian Research Data Commons (ARDC).

References

[1] 2023. mypy - Optional Static Typing for Python. https://web.archive.org/web/20230118003508/https://mypy-lang.org/.

[2] Android. 2024. SDK Platform release notes. https://developer.android.com/tools/releases/platforms.

[3] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotić, and Bernhard Scholz. 2022. 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* (Tbilisi, Georgia). Springer-Verlag, Berlin, Heidelberg, 83–102. https://doi.org/10.1007/978-3-031-16767-6_5

[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

[5] Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing-Chi Cheung, and Chang Xu. 2024. 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*. 870–882.

[6] Zhixiang Chen, Anji Li, Neng Zhang, Jianguo Chen, Yuan Huang, and Zibin Zheng. 2024. iJTyper: An Iterative Type Inference Framework for Java by Integrating Constraint- and Statistically-based Methods. arXiv:2402.09995 [cs.SE] https://arxiv.org/abs/2402.09995

[7] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1982–1993. https://doi.org/10.1145/3510003.3510061

[8] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets - Artifact. https://doi.org/10.5281/zenodo.5843327. Version v0.0.1.

[9] Yiwen Dong, Zhenyang Xu, Yongqiang Tian, and Chengnian Sun. 2025. Beyond Memorization: Evaluating the True Type Inference Capabilities of LLMs for Java Code Snippets. arXiv:2503.04076 [cs.SE] https://arxiv.org/abs/2503.04076

[10] Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models. *arXiv preprint arXiv:2308.08493* (2023).

[11] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[12] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xin Peng, Xiwei Xu, and Qinghua Lu. 2023. FQN Inference in Partial Code by Prompt-tuned Language Model of Code. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 31 (dec 2023), 32 pages. https://doi.org/10.1145/3617174

[13] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. 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* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 79, 13 pages. https://doi.org/10.1145/3551349.3556912

[14] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[15] Azmain Kabir, Shaowei Wang, Yuan Tian, Tse-Hsun (Peter) Chen, Muhammad Asaduzzaman, and Wenbin Zhang. 2024. ZS4C: Zero-Shot Synthesis of Compilable Code for Incomplete Code Snippets using LLMs. *ACM Trans. Softw. Eng. Methodol.* (Nov. 2024). https://doi.org/10.1145/3702979 Just Accepted.

[16] R. S. Malik, J. Patra, and M. Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 304–315.

[17] Meta. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/.

[18] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. arXiv:2410.05229 [cs.LG] https://arxiv.org/abs/2410.05229

[19] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[20] OpenJDK. 2024. JDK Project. https://openjdk.org/projects/jdk/.

[21] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. 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* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 440–452. https://doi.org/10.1145/3650212.3652140

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

[23] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 632–642. https://doi.org/10.1145/3180155.3180230

[24] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715

[25] Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*. Elsevier Science Inc., USA.

[26] C M Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy. 2020. 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* (San Diego, California) *(ASE '19)*. IEEE Press, 243–254. https://doi.org/10.1109/ASE.2019.00032

[27] Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, and Eneko Agirre. 2023. Did ChatGPT cheat on your test? https://hitz-zentroa.github.io/lm-contamination/blog

[28] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. 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* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 102–106. https://doi.org/10.1145/3639476.3639764

[29] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

[30] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–652. https://doi.org/10.1145/2568225.2568313

[31] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 118–129. https://doi.org/10.1145/2931037.2931058

[32] Valerio Terragni and Pasquale Salza. 2022. APIzation: Generating Reusable APIs from StackOverflow Code Snippets. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 542–554. https://doi.org/10.1109/ASE51524.2021.9678576

[33] Camilo Velázquez-Rodríguez, Dario Di Nucci, and Coen De Roover. 2023. A Text Classification Approach to API Type Resolution for Incomplete Code Snippets. *Sci. Comput. Program.* 227, C (apr 2023), 26 pages. https://doi.org/10.1016/j.scico.2023.102941

[34] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne,

Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[35] Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. 2021. Identifying Versions of Libraries used in Stack Overflow Code Snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 341–345. https://doi.org/10.1109/MSR52588.2021.00046

Received 2026-01-29; accepted 2026-06-25

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.