SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets

Yiwen Dong
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
y225dong@uwaterloo.ca

Tianxiao Gu
Alibaba Group
China
tianxiao.gu@gmail.com

Yongqiang Tian
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
yongqiang.tian@uwaterloo.ca

Chengnian Sun
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
cnsun@uwaterloo.ca

ABSTRACT

Code snippets are prevalent on websites such as Stack Overflow and are effective in demonstrating API usages concisely. However they are usually difficult to be used directly because most code snippets not only are syntactically incomplete but also lack dependency information, and thus do not compile. For example, Java snippets usually do not have import statements or required library names; only 6.88% of Java snippets on Stack Overflow include import statements necessary for compilation.

This paper proposes SnR, a precise, efficient, constraint-based technique to automatically infer the exact types 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.

We have evaluated SnR on a benchmark of 267 code snippets from Stack Overflow. 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 compile, compared to 36.0% of the import statements and 9.0% of the snippets by Coster.

CCS CONCEPTS

• **Software and its engineering** → *Automated static analysis*; **Software notations and tools**; • **Theory of computation** → *Type structures.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510061

KEYWORDS

type inference, constraint satisfaction, automated repair, datalog

ACM Reference Format:
Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510061

1 INTRODUCTION

A code snippet is a small region of source code, which is usually incomplete and thus not compilable, *e.g.*, a list of statements alone in Java without being put in a method or class. Code snippets are useful, and commonly found from well used Q&A websites like Stack Overflow (SO) where small code snippets are often used to ask questions or illustrate answers concisely.

However, when developers find a code snippet on SO to be useful and would like to use it in their projects, it is not easy to directly use the code snippet. In Figure 1, a Java compiler such as `javac` cannot resolve the types used in the snippet (*e.g.*, `Date`, `Days`, `DateFormat`) to their definitions which can be from third-party libraries. It takes developers both time and efforts to manually repair code snippets to be compilable, a challenging task which requires the developers to figure out the exact types used in the code snippet (*e.g.*, what exact types do `Date` and `DateTime` refer to) and the exact libraries that provide the definitions of the used types.

A technique to automatically repair code snippets by recovering the missing types and dependent libraries for compilation will greatly save developers time; compilable code snippets carry more semantic information than uncompilable ones and therefore such a technique can potentially enable researchers to extend existing research on SO [1, 14, 17, 21, 26–28, 41–44].

Technical Challenges. Generally, there are three technical challenges of automatically repairing and compiling code snippets.

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 fully qualified name (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

```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 1: Formatted code snippet in SO post #3329469.

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 [36] showed that only 6.88% of Java code snippets on SO included import statements that specified the FQNs of the types used in the code snippets. One example is Figure 1; the Java compiler is not able to infer the FQNs from the 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 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 FQN, 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 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 [36]; Baker extracts constraints from code snippets and uses a naive constraint solving algorithm to infer FQNs [35]; both StatType [25] and Coster [32] 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 [36], proposed constraint solving algorithm [35], or the trained statistical models [25, 32]; *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, 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, 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 suffer, SnR can deterministically, precisely infer types with explicit explanation how and why the types are inferred.

We thoroughly evaluated SnR against the state-of-the-art tool, Coster [32]. We used an established benchmark called *StatType-SO* consisting of 267 code snippets manually collected from Stack Overflow posts [25]. 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 [32]), SnR achieves high precision of 98.20% and recall of 79.66% compared to 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 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 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. This paper makes the following contributions:

- **Novelty** We proposed SnR, a novel constraint-based approach to automatically, 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.
- **Verifiability** We made a replication package available at https://doi.org/10.5281/zenodo.5843327

2 BACKGROUND

2.1 Motivating Example

We use the code snippet in Figure 1 as a motivating example to illustrate 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 1:

① It is imprecise due to its heuristic-based nature. For example, both `java.sql.Date` and `java.util.Date` are recommended

Table 1: The top-3 candidates output by SnR for Figure 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

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 [35] or more recently, statistics [25, 32]. 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 implementation 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 1, we show a sample solution by SnR for the motivating problem Figure 1. The large number of FQNs with multiple implementation (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 [39]. 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.*

```
             Expr  ::=  Name | Literal | this | Expr Op Expr
                        | ( Type ) Expr | Expr [Expr]
                        | Expr . SimpleName
                        | Expr instanceof ReferenceType
                        | Expr . SimpleName ({Expr})
                        | new ClassType ({Expr})
                        | new Type [ Expr ]
             Name  ::=  FQN | SimpleName
              FQN  ::=  Name . SimpleName
          Literal  ::=  null | NumberLiteral | StringLiteral
               Op  ::=  + | - | * | / | % | > | == | >= | !=
             Type  ::=  PrimitiveType | ReferenceType
    ReferenceType  ::=  ClassType | ArrayType
                        | ParameterizedType
    PrimitiveType  ::=  int | float | boolean
        ClassType  ::=  Name
        ArrayType  ::=  Type [ ]
ParameterizedType  ::=  ClassType <{ClassType}>
```

Figure 2: Part of a simplified grammar of the expressions in Java. {*} denotes that the enclosed term occurs zero or more times.

2.2 Java Program

This paper specifically works for Java, but the proposed methodology is generalizable to other statically-typed programming languages. This section lists minimal Java concepts which are necessary to illustrate our approach. A Java program consists of a set of compilation units each of which is a Java source file and defines one class along with any number of inner classes.

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.

Interfaces Each class can have a set of *interfaces* as supertypes.

Annotations Each class can have a set of *annotations*.

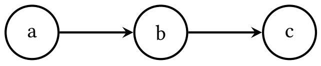
Fields Each class can have a set of *fields*. Each field has a type and a name, and can be optionally initialized with an expression.

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 shows a simplified version of the grammar of expressions in Java. We will use this grammar and the language constructs listed above to illustrate how SnR infers required types and libraries for incomplete Java code snippets.

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

(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.



(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) Datalog program

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

(d) Datalog program input representing the dependency graph

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

(e) Successful query for the reachable rule given the input in Figure 3d

Figure 3: Datalog grammar and a Datalog example program.

2.3 Datalog

Our technique leverages Datalog, a declarative logic programming language which emerged from database systems in 1980s [34]. In recent years, Datalog has found use in a whole range of applications [9, 12] in particular program analysis [3, 5, 8, 20].

A simplified version of the Datalog grammar is shown in Figure 3a. 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 3b–3e use a dependency graph as an example to illustrate the various facets of a Datalog program. The dependency graph in Figure 3b 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 3d. 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 3c. When we query for the `reachable` relation, all the reachable node pairs are returned by a Datalog solver as seen in Figure 3e, *i.e.*, `(a,b)`, `(a,c)`, and `(b,c)`.

3 METHODOLOGY

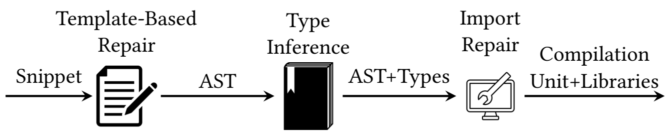


Figure 4: The overall workflow of SnR to repair a code snippet to a compilable compilation unit.

Figure 4 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 (*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 forms a syntactically valid compilation unit. Our approach is similar to that outlined by Terragni et al. [36]. 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 compiles 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 5 describes the internal components in the type inference engine. In the remainder of this section, we describe these components in detail.

3.1 Knowledge Base

Given a set of libraries, we build the knowledge base which can be queried to resolve ambiguities (§3.3) when gathering constraints and solving constraints (§3.4).

3.1.1 Content. A simplified schema for our knowledge base is described in Figure 6. 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 §3.4.

Note that the schema in Figure 6 is simplified to ease presentation with the assumption that there are no multiple classes with the same

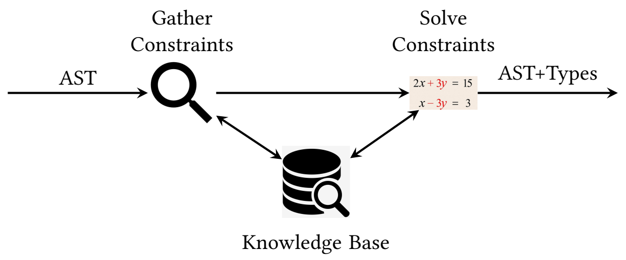


Figure 5: Type Inference Process in SnR.

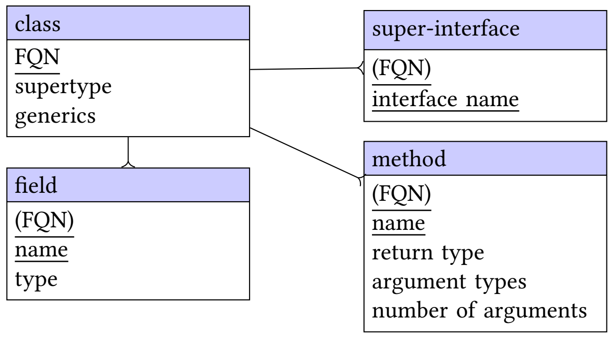


Figure 6: Simplified version of the knowledge base schema assuming there are no classes with the same FQN. Each box represents a table with a table name (in blue) and a number of column names. The underline denotes the primary key or keys that uniquely identify a row of data. The column names in parenthesis are foreign keys which are linked to primary keys in another table. An edge $\multimap$ represents a one-to-many relationship between the two connected tables.

FQN. Moreover, library information (*e.g.*, which library defines a type) is also omitted in Figure 6. The real knowledge base in SnR handles both with additional database table columns.

3.1.2 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 2 lists the query functions used in this paper.

Table 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

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"`).

3.2 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* defines as below,

Definition 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 [25, 32] 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, method calls. For high-precision type inference, SnR infers not only explicitly used types 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 paper.

3.2.1 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()`

3.2.2 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 lists all the types of constraints used in SnR, and Table 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 4 specifies that for a class declaration c, SnR generates at least two type variables (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 uses of the `subtype` constraint in order to model implicit type conversions of both primitive and reference

Table 3: Constraints used in this paper. Each constraint specifies some property that a type variable τ should have, or some 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, $\vec{\tau_{\text{arg}}}$, τ_{ret})	τ has a method with given name, argument types $\vec{\tau_{arg}}$ and return type τ_{ret}
paramtype(τ, $\vec{\tau_{\text{arg}}}$, τ_{param})	τ with parameter $\vec{\tau_{arg}}$ 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(τ, τ_{super})	τ extends τ_{super}
interface(τ, $\tau_{\text{interface}}$)	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}

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 4, SnR creates the following constraints over the type variables τ_1, τ_2, τ_3 and τ_4 of the example statement in §3.2.1.

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 Type Inference: Resolving Ambiguities

From the constraint generation rules laid out in Table 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, 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) which 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 through 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 look up $\Omega_{simplename}$(`"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.

3.4 Type Inference: Solving Constraints

Given a code snippet *s*, the set *C* of the constraints extracted from *s* by the rules in Table 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 of 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.

3.4.1 *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(τ_1, "List")	paramtype(τ_1, τ_2, τ_4)
simplename(τ_2, "Date")	subtype(τ_4, τ_3)
simplename(τ_3, "ArrayList")	method(τ_4, "get", ["int"], τ_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 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 $\Omega_{simplename}$

Table 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	$cls\ c\ ext\ t_1\ impl\ \vec{t_2}$	$c : \tau, t_1 : \tau_1, \vec{t_2} : \vec{\tau_2}$	extend(τ, τ_1) $\forall \tau_i \in \vec{\tau_2}$, interface$(\tau, \tau_i)$
Type	Type	t	$t : \tau$	simplename(τ, t)
	Array Type	$t\ []$	$t : \tau_1, \Gamma(t[]) = \tau_2$	arraytype(τ_1, τ_2)
	Paramed Type	$t_1 \langle \vec{t_2} \rangle$	$t_1 : \tau_1, \vec{t_2} : \vec{\tau_2}, \Gamma(t_1\langle \vec{t_2} \rangle) = \tau_3$	paramtype$(\tau_1, \vec{\tau_2}, \tau_3)$
Statement	If	if (e) $\{\vec{s}\}$	$\Gamma(e) : \tau$	subtype$(\tau,$ "boolean"$)$
	While	while (e) $\{\vec{s}\}$	$\Gamma(e) : \tau$	subtype$(\tau,$ "boolean"$)$
Expression	Assignment	$e_1\ =\ e_2$	$\Gamma(e_1) = \tau_1, \Gamma(e_2) = \tau_2$	subtype(τ_1, τ_2)
	Annotation	$@\ t$	$t : \tau$	annotation(τ)
	Inner Class	$t\ .\ ct$	$t : \tau_1, \Gamma(t.ct) = ct : \tau_2$	simplename(τ_2, ct) innerclass(τ_1, τ_2)
	Qualified Name	$n\ .\ sn$	$sn : \tau$	fqn$(\tau, n.sn)$
	Field	$e\ .\ f$	$\Gamma(e) = \tau_1, \Gamma(e.f) = \Gamma(f) = \tau_2$	field(τ_1, f, τ_2)
	Method	$e_1\ .\ m\ (\vec{e_2})$	$\Gamma(e_1) = \tau_1, \Gamma(\vec{e_2}) = \vec{\tau_2}, \Gamma(e_1.m(\vec{e_2})) = \tau_3$ $\vec{\tau_4} =$ [create τ_p for τ_s in $\vec{\tau_2}$]	method$(\tau_1, m, \vec{\tau_4}, \tau_3)$ $\forall \langle \tau_s, \tau_p \rangle \in \langle \vec{\tau_2}, \vec{\tau_4} \rangle$, subtype$(\tau_p, \tau_s)$
	New Instance	new $t(\vec{e})$	$t : \tau_1, \Gamma(\vec{e}) = \vec{\tau_2}, \Gamma(\text{new } t(\vec{e})) = \tau_3$ $\vec{\tau_4} =$ [create τ_p for τ_s in $\vec{\tau_2}$]	method$(\tau_1,$ "<init>"$, \vec{\tau_4}, \tau_3)$ $\forall \langle \tau_s, \tau_p \rangle \in \langle \vec{\tau_2}, \vec{\tau_4} \rangle$, subtype$(\tau_p, \tau_s)$
	Instance Of	e instanceof t	$t = \tau_1, \Gamma(e \text{ instanceof } t) = \tau_2$	subtype$(\tau_2,$ "boolean"$)$
	Array Access	$e_1[e_2]$	$\Gamma(e_1) = \tau_1, \Gamma(e_2) = \tau_2$	subtype$(\tau_2,$ "int"$)$

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(τ_1, τ_2, τ_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 subtype(τ_4, τ_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.

3.5 Type Inference: Candidate Prioritization

Given a code snippet s and the set of type variables T extracted from s, the type inference engine in §3.4 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

[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.

Table 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

is a set of triples in the form

$\{\langle \tau, fqn, lib\rangle \mid \tau \in T, fqn$ is a FQN for τ, lib is a library defining $fqn\}$

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 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 [36].

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 [32]; thus in this paper we only compare SnR with Coster. The source code of Coster, along with their model, was obtained from its github repository [29].

We use precision, recall, and F_1 score 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 [25, 32]. The dataset consists of 267 snippets from six popular libraries. Table 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. [25].

Implementation. SnR is a single-threaded Java application and uses MariaDB to serve as the knowledge base. The constraints are solved using Soufflé [13] Datalog solver. We use 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.

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 [32]) in StatType-SO. The results are summarized in Table 6. SnR significantly outperforms Coster in precision (98.20% vs 66.35%) and recall (79.66% vs 66.35%). Coster often incorrectly recommend 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 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 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 3.1) 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 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 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, reliably repair incomplete code snippets.

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 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.

Table 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%)

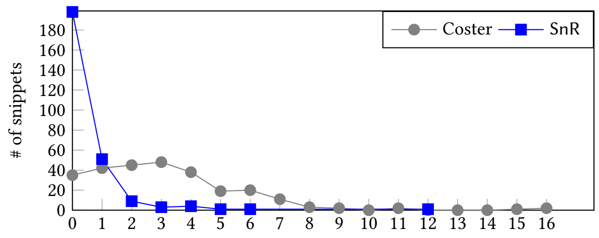


Figure 7: The distribution of code snippets *w.r.t.* number of missing import statements after repair using SnR and Coster. Points with $y = 0$ are not plotted.

Different from Table 8 which shows the information on completed import statements, Figure 7 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 [31] for finding FQNs in code snippets. The author produced a demonstration video [30] illustrating the new integration with Eclipse to fix import statements. In the video, the authors attempted to repair the code snippet from Stack Overflow post [6] 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.

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 greedy until all the missing libraries are satisfied, to validate whether our candidate prioritization heuristic detailed in §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 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

Table 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 §3.5 is effective in improving the accuracy of type inference.

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

4.4.1 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 allows for more information to be recovered from each code snippet.

4.4.2 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 8, 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.

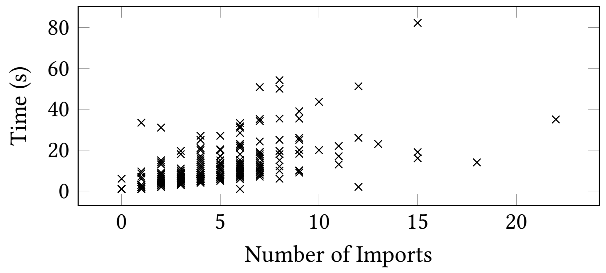


Figure 8: The time it takes to repair a snippet with *X* number of imports. Each × represents a snippet.

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.

5 DISCUSSION

In this section, we will discuss potential applications of our work (§5.1), along with the limitations of our technique (§5.2) and potential threats to validity (§5.3).

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 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 [31]. As demonstrated in §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 SO, 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 [19]. 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 [4, 21, 27, 41, 44]. 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 metric on what makes a good code snippet [21]; to aid training of algorithms that use SO snippets [41]; to better existing IDE SO code recommendation tools [27].

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.

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.

6 RELATED WORK

Type Inference. The area of type inference have a rich history in programming languages specially object-oriented languages [10, 22, 23, 33, 38]. Constraints have also been used for type analysis [2, 38]. 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 [25, 32, 35]. Our work is similar to some of the earlier works from Subramanian et al. [35] in the use of constraints but differ in some key ways such as, (1) generic type handling, (2) method of solving, and (3) selection of multiple compatible candidates. Their tool Baker relied on simple constraints and does not provide library recommendations. More recent works surpassed Baker's performance using statistical models to improve inference accuracy [25, 32]. These models are trained using a large set of existing [18] or collected popular Github projects and are evaluated using hand collected SO posts including StatType-SO. Saifullah et al. [32] improved upon the model by Phan et al. [25] by leveraging local and global context. Our work improves upon the state of the art and compliment the existing techniques. Future techniques can incorporate the accuracy of constraint based techniques with the performance of statistics based ones.

Deep learning models [11, 16] has been used to conduct type inference on dynamically typed languages. Unlike SnR, these techniques often use additional sources of information such as comments, method and variable naming to conduct their inference.

Partial Program Analysis. RecoDoc [7] 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 have leveraged SO snippets to build better development tools [27, 41], to study usages in open source projects [4, 17, 43, 44], and more [25, 32, 36, 40]. Recent work have used heuristics based approaches to automatically synthesis compilable code snippets [36, 37]. 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 [19], semantic errors leveraging test cases [15], or specifications (pre- and postconditions) [24]. Our work focuses on code snippets which are incomplete code, without tests or specifications in most cases. Our technique need to be more flexible and cannot rely on having test code or specifications.

7 CONCLUSION

This paper proposes 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 of boosting developers' productivity with code snippets.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their insightful comments. We would like to thank the authors of StatType for providing their benchmark and authors of COSTER for helping us run their tool. This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant.

REFERENCES

[1] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. 2017. On code reuse from StackOverflow: An exploratory study on Android apps. *Information and Software Technology* 88 (2017), 148–158. https://doi.org/10.1016/j.infsof.2017.04.005

[2] Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 31–41. https://doi.org/10.1145/165180.165188

[3] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. 2015. 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* (Portland, OR, USA) *(SOAP 2015)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/2771284.2771287

[4] Sebastian Baltes and Stephan Diehl. 2019. Usage and attribution of Stack Overflow code snippets in GitHub projects. *Empirical Software Engineering* 24, 3 (2019), 1259–1295.

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[6] cianBuckley. 2013. java - Joda Time converting time zoned date time to milliseconds - Stack Overflow. Retrieved December 22, 2020 from https://web.archive.org/web/20170227042935/http://stackoverflow.com/questions/18274902/jodatime-converting-time-zoned-date-time-to-millis

[7] Barthélémy Dagenais and Martin P. Robillard. 2012. Recovering Traceability Links between an API and Its Learning Resources. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 47–57.

[8] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. 1996. 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* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/231379.231399

[9] Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. 2012. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Vol. 6702. Springer.

[10] David Greenfieldboyce and Jeffrey S. Foster. 2007. Type Qualifier Inference for Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 321–336. https://doi.org/10.1145/1297027.1297051

[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] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1213–1216. https://doi.org/10.1145/1989323.1989456

[13] 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.

[14] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 664–675. https://doi.org/10.1145/2568225.2568292

[15] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[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] S. S. Manes and O. Baysal. 2019. 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)*. 235–239. https://doi.org/10.1109/MSR.2019.00047

[18] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: A Dataset of Compilable, and Compiled, Java Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3196398.3196450

[19] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Eddie Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors.

[20] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[21] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. 2012. What makes a good code example?: A study of programming Q A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 25–34. https://doi.org/10.1109/ICSM.2012.6405249

[22] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. 1992. Making type inference practical. In *ECOOP '92 European Conference on Object-Oriented Programming*, Ole Lehrmann Madsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–349.

[23] Jens Palsberg and Michael I. Schwartzbach. 1991. Object-Oriented Type Inference. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, USA) *(OOPSLA '91)*. Association for Computing Machinery, New York, NY, USA, 146–161. https://doi.org/10.1145/117954.117965

[24] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[25] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 632–642.

[26] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack overflow in the ide. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1295–1298.

[27] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 102–111.

[28] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto. 2019. Toxic Code Snippets on Stack Overflow. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2900307

[29] C. M. K. Saifullah. 2020. COSTER. Retrieved May 18, 2020 from https://github.com/khaledkucse/COSTER

[30] C. M. K. Saifullah. 2020. COSTER: A Tool for Finding Fully Qualified Names of API Elements in Online Code Snippets. Retrieved December 22, 2020 from https://youtu.be/oDZtw9MzUWM?t=208

[31] C M Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal Roy. 2021. COSTER: A Tool for Finding Fully Qualified Names of API Elements in Online Code Snippets *(ICSE '21 DEMO)*.

[32] C. M. K. Saifullah, M. Asaduzzaman, and C. K. Roy. 2019. Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 243–254.

[33] Daniel Smith and Robert Cartwright. 2008. 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* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 505–524. https://doi.org/10.1145/1449764.1449804

[34] Michael Stonebraker. 1988. *Readings in database systems*. Morgan Kaufmann Publishers Inc.

[35] 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

[36] 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

[37] Valerio Terragni and Pasquale Salza. 2021. APIzation: Generating Reusable APIs from StackOverflow Code Snippets. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 542–554. https://doi.org/10.1109/ASE51524.2021.9678576

[38] Tiejun Wang and Scott F. Smith. 2001. Precise Constraint-Based Type Inference for Java. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–117.

[39] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 319–330. https://doi.org/10.1145/3236024.3236056

[40] A. W. Wong, A. Salimi, S. Chowdhury, and A. Hindle. 2019. 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)*. 318–322.

[41] E. Wong, Jinqiu Yang, and Lin Tan. 2013. AutoComment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 562–567. https://doi.org/10.1109/ASE.2013.6693113

[42] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/2901739.2901767

[43] D. Yang, P. Martins, V. Saini, and C. Lopes. 2017. Stack Overflow in Github: Any Snippets There?. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 280–290. https://doi.org/10.1109/MSR.2017.13

[44] T. Zhang, D. Yang, C. Lopes, and M. Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 316–327. https://doi.org/10.1109/ICSE.2019.00046