Slicing Based Code Recommendation for Type Based Instance Retrieval

Rui Sun, Hui Liu(✉), and Leping Li

Beijing Institute of Technology, Beijing, China
{sr1993,liuhui08,lileping}@bit.edu.cn

Abstract. It is common for developers to retrieve an instance of a certain type from another instance of other types. However, it is quite often that developers do not exactly know how to retrieve the instance although they know exactly what they need (the instance to re retrieved, also known as the *target instance*) and where it could be retrieved (i.e., the *source instance*). Such kind of instance retrieval is popular and thus their implementations, in different forms, are often publicly available on the Internet. Consequently, a number of approaches have been proposed to retrieve such implementations (code snippets) and release developers from reinventing such snippets. However, the performance of such approaches deserves further improvement. To this end, in this paper, we propose a slicing based approach to recommending code snippets that could retrieve the target instance from the source instance. The approach works as follows. First, from a large code base, it retrieves methods that contain the source instance and the target instance. Second, for each of these methods, it locates the target instances, and extracts related code snippets that generate the target instances by backward code slicing. Third, from the extracted code snippets, it removes those that do not contain the source instance. Fourth, it merges code snippets whose corresponding target instances are at parallel execution paths. Fifth, it removes duplicate code snippets. Finally, it ranks the resulting code snippets, and presents the top ones. We implement the approach as an Eclipse plugin called *TIRSnippet*. We also evaluate it with real type based instance retrieval queries. Evaluation results suggest that compared to the state-of-the-art approaches, the proposed approach improves the precision and recall by 8.8%, and 25%, respectively.

Keywords: Reuse · Slicing · Code search

1 Introduction

It is common for developers to retrieve an instance of a certain type from another instance they have [12]. For convenience, we call the retrieve Type Based Instance Retrieval (TIR). For example, they may want to retrieve the *IDocument* instance associated with an *IFile* instance. The following is a typical code snippet that accomplishes the task:

© Springer Nature Switzerland AG 2019

```
IFile file = ...
IPath path= file.getFullPath();
ITextFileBufferManager bufferManager = FileBuffers.getTextFileBufferManager();
bufferManager.connect(path, LocationKind.IFILE, null);
ITextFileBuffer textFileBuffer = bufferManager.getTextFileBuffer(path, LocationKind.
     IFILE);
IDocument document = textFileBuffer.getDocument();
```

For convenience, code snippets (like the example) that accomplish a TIR are noted as TIR code snippets. The instance (*document* in the example) that a TIR code snippet returns is called *target instance*, and the input (*file* in the example) is called *source instance*.

Automatic or semi-automatic recommendation of TIR code snippets is desirable. TIR code snippets could be quite complicated. Consequently, it is challenging for programmers who are not familiar with relevant framework or library to create such a code snippet from scratch. However, it is likely that the programming task has been implemented by others and such implementation (TIR code snippet) is available on the Internet. Retrieving such reference implementation for reuse (or at least as learning material) can significantly facilitate the programming task at hand [9]. To this end, a number of approaches have been proposed [12,17,20] to retrieve TIR code snippets according to their target instances and source instances. Mandelin et al. [12] proposed *PROSPECTOR*. It accepts a query in the form of (*Tin*, *Tout*), where *Tin*, *Tout* represent two different types. *PROSPECTOR* utilizes API method signatures and a corpus of examples to synthesize API client code automatically which achieves the transformation from *Tin* to *Tout*. Shavechaphan and Claypool [17] use a graph-based module to mine for paths for the instantiation of a certain type. These paths form suggested code snippets which are extracted from a sample repository. Thummalapenta and Xie [20] use a code search engine to collect relevant code samples, and extract method-invocation sequences that retrieve the target instance by static code analysis. Such approaches [12,17,20] have greatly facilitated the reuse of type based instance retrieval code snippets. However, the performance (e.g., precision and recall) of such approaches deserve significant improvements.

To this end, in this paper, we propose a slicing based approach to recommending TIR code snippets. For a given TIR query in the form of <*source type*, *target type*>, the proposed approach retrieves code snippets that accomplish the given TIR query. The approach works as follows. First, from a code repository, it retrieves methods that contain both source instance and target instance. Second, for each such method, it employs code slicing to extract code snippets that generate target instances. Third, it merges extracted code snippets whose corresponding target instances are at parallel execution paths. Fourth, it removes duplicate code snippets, and ranks the resulting code snippets to recommend the top ones. We also implement the proposed approach as an Eclipse plugin, called *TIRSnippet*.

The paper makes the following contributions:

- A slicing based approach to recommending code snippets for TIR queries.
- Implementation of the proposed approach as an Eclipse plugin.
- Evaluation of the proposed approach with real TIR queries. Evaluation results suggest that the proposed approach improves the state-of-the-art.

The rest of the paper is structured as follows. Section 2 presents a motivating example. Section 3 presents details of the proposed approach. Section 4 presents an evaluation of the proposed approach. Section 5 provides a review of related research. Section 6 makes conclusions.

2 Motivating Example

Suppose that we are assigned the following programming task: For a given method invocation, retrieve the type of the instance on which the method is invocated. For example, given the method invocation '*assign.getOperator*()', we should retrieve the type of *assign*. If we decide to accomplish this task with JDT framework [2] (a widely used Java development toolkit), we should retrieve an *ITypeBinding* instance (initializing the data type of an instance) from a given *MethodInvocation* instance. An example code snippet (we call it *ExampleSnippet*) that can accomplish this task is presented as follows:

```
1        Expression expression = invocation.getExpression();
2        if(expression == null){
3         typeBinding = invocation.resolveMethodBinding().getDeclaringClass();
4        }
5        else {
6         typeBinding = expression.resolveTypeBinding();
7        }
```

where *invocation* on Line 1 is a *MethodInvocation* instance and *typeBinding* on Line 3 and Line 6 is an *ITypeBinding* instance.

We expect existing TIR approaches to retrieve the *ExampleSnippet* for the given task. However, these approaches, e.g., *PARSEWeb* [20], fail. *PARSEWeb* creates a graph for the *ExampleSnippet* where nodes represent statements (method invocation, constructor or typecast statement) and edges represent control information between statements. Based on the graph, *PARSEWeb* searches for paths that connect the source instance (*invocation* in the example) and the target instance (*typeBinding* in the example). For each of the resulting paths, statements on the path that are relevant to the target instance make up a TIR code snippet. Consequently, code snippet 1# and code snippet 2# (as presented in Fig. 1) are retrieved. However, both of the retrieved code snippets are incomplete and may result in buggy implementation. For example, code snippet 1# may result in *Null Pointer Exception* during execution if a *MethodInvocation* instance has no expression (e.g., for method invocation '*getFullName()*').

```
Code Snippet 1#
Expression expression = invocation.getExpression();
typeBinding = expression.resolveTypeBinding();

Code Snippet 2#
typeBinding = invocation.resolveMethodBinding().getDeclaringClass();
```

Fig. 1. Code snippets retrieved by existing approaches

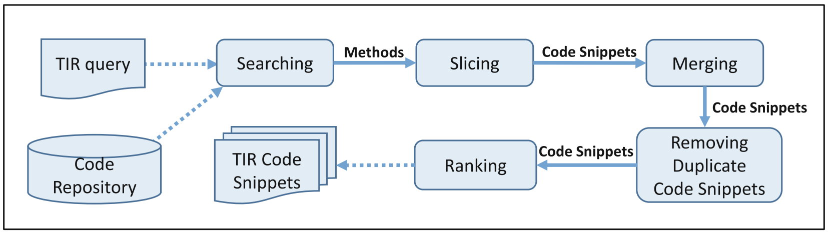


Fig. 2. Overview of the proposed approach

In contrast, the proposed approach retrieves the complete *ExampleSnippet* successfully. It first locates target instances (*typeBinding* on Line 3 and Line 6 in the example). After that, for each target instance, it employs code slicing to extract all statements (including complex control statements) that are relevant to the generation of the target instance. In the example, two code snippets are extracted. They are noted as TIRC (*typeBinding*, Line 3) = {1, 2, 3, 4} and TIRC (*typeBinding*, Line 6) = {1, 2, 4, 5, 6, 7} where the integer set represents line numbers, and statements on these lines make up the extracted TIR code snippets. Unlike code snippet 1# and code snippet 2# retrieved by existing approaches, such code snippets retrieved by the proposed approach contain control statements as well. The proposed approach notices that the two target instances on Line 3 and Line 6 are at parallel execution paths, and thus it merges their corresponding TIR code snippets: TIRC (*typeBinding*)={1, 2, 3, 4, 5, 6, 7}. As a result, the whole code snippet (*ExampleSnippet*) is retrieved by the proposed approach.

3 Approach

3.1 Overview

An overview of the proposed approach is presented in Fig. 2. The input of the proposed approach is a TIR query in the form of <*source type*, *target type*>. The output is the resulting TIR code snippets that could retrieve the target instance associated with the given source instance. For a TIR query, the proposed approach works as follows:

- First, from a code repository, it retrieves all methods which contain both source instance and target instance.

- Second, for each of these methods, it locates the target instances, and extracts related code snippets that generate the target instances by backward code slicing. From extracted code snippets, it removes those that do not contain the source instance.
- Third, it merges extracted code snippets whose corresponding target instances are at parallel execution paths of the involved programs.
- Fourth, from the resulting code snippets, it removes the duplicate ones.
- Finally, it ranks the resulting code snippets, and present the top ones.

3.2 Searching

The first step of the proposed approach is to search for all methods (noted as RMS) that contain both source instance and target instance from a code repository. Algorithm 1 shows the details of how to search for these methods. For each source code file in a code repository, we retrieve all methods contained in this file (Lines 2–3). For each retrieved method (noted as rm), we check whether this method contains both source instance and target instance. If yes, the method rm is added to RMS. The checking process is explained as follows. First, we retrieve all instances (noted as $INSES$) contained in the method rm (Line 5). Second, for each instance in $INSES$, we check whether it is a source instance or target instance (Lines 8–14). Finally, the method rm is added to RMS if it contains both source instance and target instance (Lines 15–16).

3.3 Slicing

From retrieved methods RMS which contain both source instance and target instance, backward code slicing is employed to extract the smallest code snippets that could generate the target instances. The slicing works as follows. First, for each method in RMS, we locate all target instances. Second, based on each of the target instances, we extract code snippet that could generate the target instance by backward code slicing. Third, from extracted code snippets, we remove those that do not contain the source instance. We take code snippet *ExampleSnippet* (as presented in Sect. 2) as an example to illustrate the detailed process. *ExampleSnippet* is a solution for the TIR query <*MethodInvocation*, *ITypeBinding*>. From this code snippet, the proposed approach first locates two target instances. These two target instances (noted as $t3$ and $t6$) are on Line 3 and Line 6, respectively. Based on target instances $t3$ and $t6$, backward code slicing is employed to extract two code snippets that could generate $t3$ and $t6$, respectively. Code snippet 3# (as presented in Fig. 3) is extracted based on target instance $t3$ and code snippet 4# (as presented in Fig. 3) is extracted based on target instance $t6$. These two code snippets both contain the source instance $invocation$, and thus we do not remove any of them.

In some special cases, slicing could not retrieve some TIR code snippets. These code snippets have the following two features: (1) Their source instances or target instances are not explicitly denoted (e.g., the source

Algorithm 1. Searching for Methods that Contain Source Instance and Target Instance

```
Input: $sourceFiles$ //source code files in a code repository
       $source\ type$ //source type in a query
       $target\ type$ //target type in a query
Output: $RMS$ //methods that contain source instance and target instance
1: $RMS \leftarrow \emptyset$
2: for each $source$ in $sourceFiles$ do
3:     $allMethods \leftarrow source$.getMethods() //retrieve methods of a source code file
4:     for each $method$ in $allMethods$ do
5:         $INSES \leftarrow method$.getInsideInstances() //retrieve all instances
6:         $containSource \leftarrow$ False
7:         $containTarget \leftarrow$ False
8:         for each $instance$ in $INSES$ do
9:             if $instance$.getType().equals($sourceType$) then
10:                $containSource \leftarrow$ True
11:            end if
12:            if $instance$.getType().equals($targetType$) then
13:                $containTarget \leftarrow$ True
14:            end if
15:            if $containSource$ == True && $containTarget$ == True then
16:                $RMS$.add($method$)
17:                break
18:            end if
19:        end for
20:    end for
21: end for
22: return $RMS$
```

instance is the instance returned by a method invocation). (2) They consist of method invocation sequences only. We take the following example to explain how to retrieve these code snippets. Suppose that we have a code snippet $PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage()$ (noted as MI), and we want to retrieve an *IWorkbenchWindow* instance from an *IWorkbench* instance. To achieve that, we first list all instances contained in MI from left to right. These instances are $PlatformUI$, $IWorkbench$, $IWorkbenchWindow$, and $IWorkbenchPage$ instance. The last three instances are listed because they are the instances returned by method invocation $getWorkbench()$, $getActiveWorkbenchWindow()$ and $getActivePage()$, respectively. After that, we check whether the source instance appears ahead of the target instance. If yes, a correct TIR code snippet is found. In this example, the source instance (the $IWorkbench$ instance) appears ahead of the target instance (the $IWorkbenchWindow$ instance). Consequently, a correct TIR code snippet

```
Code Snippet 3#
Expression expression = invocation.getExpression();
If(expression == null){
      typeBinding = invocation.resolveMethodBinding().getDeclaringClass();
}

Code Snippet 4#
Expression expression = invocation.getExpression();
If(expression == null){

}
else {
      typeBinding = expression.resolveTypeBinding();
}
```

Fig. 3. Extracted code snippets based on different target instances

$PlatformUI.getWorkbench().getActiveWorkbenchWindow()$ is retrieved. In this TIR code snippet, the source instance is the instance returned by the method invocation $PlatformUI.getWorkbench()$ and the target instance is the instance returned by the method invocation $getActiveWorkbenchWindow()$.

3.4 Merging

Extracted TIR code snippets that satisfy the following conditions are merged: (1) First, they are from the same method of a source code file; (2) Second, their corresponding target instances are at parallel execution paths of the involved program. Each of such snippets presents a special condition where the target instance could be generated. To guarantee that a target instance could be successfully generated under different conditions (i.e., handling some exceptional cases), we merge code snippets whose corresponding target instances are at parallel execution paths (i.e., these target instances could not be generated with one execution path). For example, the target instances generated by code snippet 3# and code snippet 4# (as presented in Fig. 3) are at parallel execution paths, and they are generated under different conditions. When a *MethodInvocation* instance has no expression (e.g., '*getFullName()*'), code snippet 3# is used to generate the target instance. When a *MethodInovcation* instance contains an expression (e.g., '*var.getFullName()*'), code snippet 4# is used to generate the target instance. We merge code snippet 3# and code snippet 4# into a complete code snippet (*ExampleSnippet* as presented in Sect. 2) to guarantee that a target instance could be generated no matter whether a *MethodInvocation* instance contains an expression or not. After merging, we get a set of TIR code snippet candidates (noted as $snippetCandidates$).

3.5 Removing Duplicate Code Snippets

From TIR code snippet candidates $snippetCandidates$, we remove the duplicate or nearly duplicate ones. When users look through the top k recommended TIR code snippets, the duplicate ones are meaningless and redundant. To remove these duplicates, we first make some pretreatments for each TIR code snippet in *snippetCandidates*. These pretreatments include: (1) Replacing all the variable names with a special character $'\#'$. (2) Removing blank characters from all statements. These pretreatments are conducted in order. After that, we compare any pair of TIR code snippets (noted as *snippetA* and *snippetB*) to check whether they are duplicate. To avoid comparing twice for a pair of code snippets, we guarantee that the index of *snippetA* is not larger than that of *snippetB* in the *snippetCandidates*. *SnippetA* and *snippetB* are considered as duplicate if they meet the following two conditions. The first condition is that they have the same number of statements. The second condition is that they share the same characters for all statements. If *snippetA* and *snippetB* are duplicate code snippets, we remove *snippetA* that has a smaller index.

3.6 Ranking

We rank TIR code snippet candidates *snippetCandidates*, and present the top ones. Existing investigation suggested that users usually look through only the first around ten recommendations of search or mining process [3]. Consequently, we rank *snippetCandidates* as the following process. First, we rank them by frequency. Frequency is selected for the following reasons. (1) Generally speaking, a specific function could be implemented by various approaches. (2) An approach has a higher frequency in programs, which suggests that the approach is generally acceptable and more standardized. Second, we rank *snippetCandidates* by Mccabe Cyclomatic Complexity (MCC) [13] in descending order if two TIR code snippets share the same frequency.

4 Evaluation

In this section, we present the evaluation of the proposed approach on 26 real type based instance retrieval requests post on *Stackoverflow*.

4.1 Research Questions

- **RQ1:** Are type based instance retrieval code snippets popular?
- **RQ2:** Can the proposed approach outperform the state-of-the-art approaches in retrieving TIR code snippets?
- **RQ3:** How does the code slicing influence the performance of the proposed approach?
- **RQ4:** How does the ranking influence the performance of the proposed approach?

The proposed approach is based on the assumption that type based instance retrieval code snippets are popular (i.e., it is common for developers to write type based instance retrieval code snippets). If not, the proposed approach will not be used frequently, and thus useless. Answering research question RQ1 helps to validate the assumption.

RQ2 concerns the performance of the proposed approach against the-state-of-art approaches. To answer RQ2, we compare the proposed approach against *PARSEWeb* [20] and *Google* (https://www.google.com). *PARSEWeb* is the state-of-the-art approach and *Google* is the state-of-the-practice general purpose search engine. We select PARSEWeb for the following reasons. First, it is designed to retrieve type based instance retrieval code snippets, just as the proposed approach is. Second, we fail to get the implementation of other approaches for type based instance retrieval. *PARSEWeb* is the only one of such approaches whose implementation is publicly available. Third, *PARSEWeb* has proved to have a better performance than other type based instance retrieval approaches (e.g., *PROSPECTOR* and *XSnippet*) [20]. *Google* is selected because of the following reasons. First, programmers tend to use the general purpose search engine to search for code to reuse [8,18,19]. Second, *Google* is the most frequently used search engine for code search [16,18]. It should be noted that the evaluation results do not suggest that the proposed approach is better in general than *Google*. The proposed approach is only confined to type based instance retrieval, while *Google* is more generic. Consequently, it is unfair to compare them in general. The purpose of our evaluation is to validate that better performance could be achieved by focusing on a special and common case, type based instance retrieval.

As specified in Sect. 3.3, the proposed approach is based on code slicing. Answering research question RQ3 helps to reveal the influence of code slicing in retrieving type based instance retrieval code snippets.

RQ4 concerns the influence of ranking. As specified in Sect. 3.6, we rank code snippets to present the top ones. Answering research question RQ4 helps to reveal whether the performance is influenced by the ranking.

4.2 Type Based Instance Retrieval Requests and Code Repository

To evaluate the proposed approach, we need some real TIR requests for evaluation. A request is called TIR request if it could be translated into a TIR query <*source type*, *target type*>. In the evaluation, we select TIR requests related to JDT framework [2]. This framework is chosen for the following reasons. First, the data types in a TIR request are various. They could be from any framework or library. Second, JDT framework is popular for developers to reuse. To get these requests, we search *Stackoverflow* (https://stackoverflow.com, one of the most popular online community for developers to ask and answer questions) with keywords like 'jdt get from', 'jdt convert from', 'jdt change to', etc. We manually check the top 50 returned requests to identify whether any request could be translated into a TIR query according to its description. For example, the following request is post on *Stackoverflow*:

"This plugin processes each Java file as a ICompilationUnit. However, in my approach I can only get an instance of IFile. How can I create a ICompilationUnit from this IFile object?"

According to the description, this request could be translated into a TIR query *<IFile, ICompilationUnit>*. Consequently, this request is a TIR request. Finally, 26 TIR requests (noted as *selectedRequests*) are selected for evaluation.

To search for solutions (TIR code snippets) for *selectedRequests*, we need a relevant code repository. In this evaluation, the code repository (noted as *CR*) is composed of 5 closed-source applications developed previously in Beijing Institute of Technology and 33 open-source applications (https://github.com/sr1993/TIRSnippet) downloaded from *Github*. All these applications are related to JDT framework. To get these open-source applications, we searched *Github* with keywords like 'jdt -language-:java', 'eclipse -language-:java', etc. We sorted returned applications by their stars in descending order, and selected applications. The selected applications meet the following criteria: (1) They are written in Java. (2) They are related to JDT framework. (3) They could be imported into Eclipse IDE successfully. The third criteria makes sure that we could create Abstract Syntax Tree successfully for the source code files, and thus retrieve the binding information of program elements. The size (LOC) of applications in the code repository *CR* varies from 136 to 545638.

4.3 RQ1: Type Based Instance Retrieval Code Snippets in Programs

To answer research question RQ1, we first acquire TIR code snippets from code repository *CR* which is used in the evaluation. After that, we classify those TIR code snippets into different categories. Two TIR code snippets are classified into the same category if they share the same source type and target type. Finally, we calculate the following metrics: the number of TIR code snippets, involved files, involved methods, TIR code snippet categories and the average number of TIR code snippets per application, per involved file, per involved method. Evaluation results are presented in Table 1.

Table 1. TIR code snippets in the code repository

Number of TIR code snippets	102,420
Number of involved files	5,225
Number of involved methods	56,905
Number of TIR code snippet categories	12,590
Average number of TIR code snippets per application	2,695
Average number of TIR code snippets per involved file	20
Average number of TIR code snippets per involved method	2

From Table 1, we make the following observations:

- First, the number of TIR code snippets is quite large. The results suggest that 102,420 TIR code snippets exist in 5,225 involved source code files and 56,905 involved methods. The average numbers of TIR code snippets per application, per involved file and per involved method are 2,695, 20 and 2, respectively.
- Second, the number of TIR code snippet categories is large. The result suggests that required TIR code snippets could be classified into 12,590 categories.

From the analysis in the preceding paragraphs, we conclude that type based instance retrieval code snippets are popular.

4.4 RQ2: Comparison Against Existing Approaches

To address RQ2, we compare the proposed approach against *PARSEWeb* and *Google* on 26 TIR requests in *selectedRequests*. The process is as follows:

- First, for each TIR request in *selectedRequests*, we translate it into a TIR query in the form of <*source type*, *target type*>.
- Second, for each TIR query, we apply each approach to search for TIR code snippets. It is noted that Google Code Search (www.google.com/codesearch) used in *PARSEWeb* is not available anymore. We replace it with Search Code (https://searchcode.com, a free source code search engine).
- Finally, for each query, we manually check the correctness of the top k code snippets recommended by various approaches. A code snippet is considered as correct if it could solve the query functionally and answer the corresponding request. The manual checking is conducted by three postgraduate students in Beijing Institute of Technology. They all have rich experience in Java development. They first check the recommended code snippets independently. After that, they discuss together to remove inconsistence.

Metrics. To evaluate the performance of various approaches, we employ metrics $p@k$ to measure precision and $r@k$ to measure recall. $p@k$ is calculated as follows: $p@k = \frac{tp@k}{retrievedN_k}$ where $tp@k$ represents the number of retrieved correct code snippets and $retrievedN_k$ represents the total number of retrieved code snippets, when the top k results are inspected. $r@k$ is calculated as follows: $r@k = \frac{solved@k}{N_{query}}$ where N_{query} represents the number of queries and $solved@k$ represents the number of solved queries, when the top k results are inspected.

We also assess the performance of various approaches with the Mean Reciprocal Rank (MRR). It is a statistical metric to evaluate a process that produces a list of possible responses to a specific query [4]. It is calculated as follows: $MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$ where $rank_i$ represents the rank of the first correct code snippet for query i and Q represents all queries. The approach with a higher MRR has a better performance.

Table 2. Comparison against existing approaches

	Proposed approach	*PARSEWeb*	*Google*
p@1	87%	80%	23.1%
p@5	68%	66.3%	30.8%
p@10	65%	58.5%	25.8%
r@1	76.9%	61.5%	19.2%
r@5	84.6%	76.9%	69.2%
r@10	84.6%	76.9%	80.8%
MRR	0.825	0.609	0.395

Results. Evaluation results are presented in Table 2. From Table 2, we make the following observations:

- First, the proposed approach is accurate. The precision is 87% when the top 1 results are inspected. Compared to *PARSEWeb* and *Google*, the proposed approach improves the precision by 8.8% = (87% – 80%)/80% and 276.6% = (87% – 23.1%)/23.1%, respectively.
- Second, the proposed approach improves recall significantly. Its recall is always greater than that of *PARSEWeb* and *Google*. When the top 1 results are inspected, it improves recall by 25% = (76.9% – 61.5%)/61.5% and 300.5% = (76.9% – 19.2%)/19.2%, compared to *PARSEWeb* and *Google* respectively.
- Third, the proposed approach is significantly more effective in ranking resulting code snippets than *PARSEWeb* and *Google*. It improves the MRR by 35.5% = (0.825 – 0.609)/0.609 and 108.9% = (0.825 – 0.395)/0.395, respectively.

Analysis. To investigate the reasons why the proposed approach has a better performance than the state-of-the-art approach *PARSEWeb*, we look through and compare code snippets retrieved by these two approaches, and we attribute the advantage of the proposed approach to the following two points:

First, the exact type information of Java elements in source code files could be acquired in the proposed approach. Consequently, some correct code snippets whose source instance or target instance is not explicitly denoted could be retrieved, e.g., the assignment (code snippet) $IType\ type = baseType.getJavaProject().findType(refTypeName, (IProgressMonitor)null);$ where an *IType* instance *type* is at the left hand of this assignment and an *IJavaProject* instance *baseType.getJavaProject*() (the return type of this method invocation is *IJavaProject*) is at the right hand of this assignment. Consequently, this code snippet is a solution for the query <*IJavaProject*, *IType*>. The proposed approach gets the type information through Abstract Syntax Tree, and thus it could acquire exact type information of any Java elements. Consequently, it could retrieve this kind of code snippet. However, *PARSEWeb* gets the type information heuristically and has no idea about the return type of the method invocation *baseType.getJavaProject*(), and thus this code snippet could not be retrieved.

Second, the proposed approach could retrieve complete code snippets. We check the code snippets retrieved by the proposed approach, and observe that all retrieved code snippets are complete. Additionally, due to code slicing and merging are employed in the proposed approach, control statements (e.g., *if*, *for* statements) could be included in the retrieved code snippets. Because of these control statements, the biggest advantage of corresponding code snippets is that they are more ready-to-use. The following code snippet is an example:

```
Iterator iter = selection.iterator();
while(iter.hasNext()){
   Object element = iter.next();
   if(element instanceof IJavaProject){
      project = (IJavaProject) element;
   }else if(element instanceof IPackageFragment){
      IPackageFragmentRoot pf = (IPackageFragmentRoot) element;
      project = pf.getJavaProject();
   }else if(element instanceof ICompilationUnit){
      ICompilationUnit cu = (ICompilationUnit) element;
      project = cu.getJavaProject();
   }else if(element instanceof IProject){
      IProject p = (IProject) element;
      if(p.isOpen() && p.hasNature(JavaCore.NATURE_ID)){
         project = JavaCore.create(p);
      }
   }
}
```

where *selection* is an *IStructuredSelection* instance and *project* is an *IJavaProject* instance.

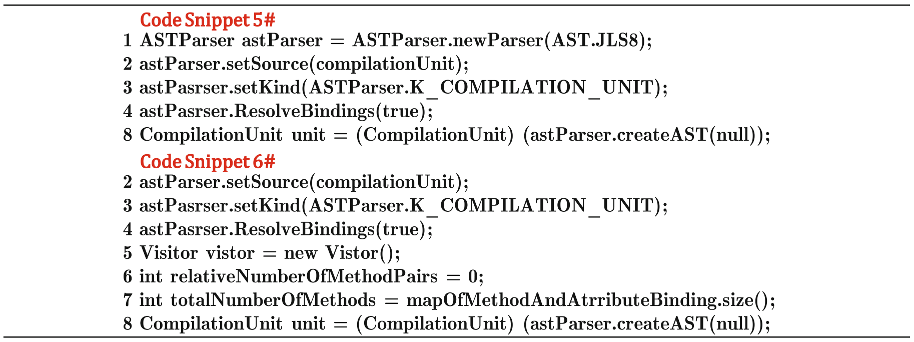
This code snippet is one of the code snippets retrieved by the proposed approach for the query <*IStructuredSelection*, *IJavaProject*>. From the code snippet, we observe it has two features. First, it is complete and could be safely reused. It contains several *if/else* statements. These statements guarantee that a target instance (*project* in the example) could be generated under different conditions. Second, it is easy to reuse. Only the name of *selection* or *project* is needed to modify if necessary. Consequently, code snippets containing control statements are more ready-to-use. We also observe that this kind of code snippets account for 30.4%, 37.1%, 32.8% of the top $k\,(k = 1, 5, 10)$ code snippets recommended by the proposed approach, respectively. Whereas, *PARSEWeb* could retrieve method invocation sequences only, and they may be difficult to reuse without needed modification.

4.5 RQ3: Influence of Code Slicing

To reveal the influence of code slicing, we first look through the code snippets (noted as *SlicedSnippets*) retrieved by the proposed approach in Sect. 4.4. After that, for each code snippet in *SlicedSnippets*, we compare it with its corresponding non-sliced snippet which is retrieved by collecting all statements between the source instance and the target instance (inclusive) from source code file. The following code snippet (we call it *SourceSnippet*) is taken as an example to illustrate what we did.

```
1 ASTParser astParser = ASTParser.newParser(AST.JLS8);
2 astParser.setSource(compilationUnit);
3 astPasrser.setKind(ASTParser.K_COMPILATION_UNIT);
4 astPasrser.ResolveBindings(true);
5 Visitor vistor = new Vistor();
6 int relativeNumberOfMethodPairs = 0;
7 int totalNumberOfMethods = mapOfMethodAndAtrributeBinding.size();
8 CompilationUnit unit = (CompilationUnit) (astParser.createAST(null));
```

The code snippet *SourceSnippet* is from a source code file, and *compilationUnit* on Line 2 is an *ICompilationUnit* instance. For a TIR query <*ICompilationUnit*, *CompilationUnit*>, the source instance *compilationUnit* is on Line 2 and the target instance *unit* is on Line 8. From *SourceSnippet*, code slicing is used in the proposed approach to retrieve code snippet 5# (as presented in Fig. 4). If code slicing is not applied, collecting statements between the source instance *compilationUnit* one Line 2 and the target instance *unit* one Line 8 (inclusive) will result in the code snippet 6# (as presented in Fig. 4).


Code Snippet 5#

```
1 ASTParser astParser = ASTParser.newParser(AST.JLS8);
2 astParser.setSource(compilationUnit);
3 astPasrser.setKind(ASTParser.K_COMPILATION_UNIT);
4 astPasrser.ResolveBindings(true);
8 CompilationUnit unit = (CompilationUnit) (astParser.createAST(null));
```

Code Snippet 6#

```
2 astParser.setSource(compilationUnit);
3 astPasrser.setKind(ASTParser.K_COMPILATION_UNIT);
4 astPasrser.ResolveBindings(true);
5 Visitor vistor = new Vistor();
6 int relativeNumberOfMethodPairs = 0;
7 int totalNumberOfMethods = mapOfMethodAndAtrributeBinding.size();
8 CompilationUnit unit = (CompilationUnit) (astParser.createAST(null));
```

Fig. 4. Sliced and non-sliced code snippets

From code snippet 5# and code snippet 6#, we make the following observations:

Table 3. Influence of ranking

	Default	Disabling *Ranking*	Disabling $C1$	Disabling $C2$
p@1	87%	60.9%	60.9%	82.6%
p@5	68%	58.8%	60.8%	68%
p@10	65%	54.4%	58.3%	59.4%
r@1	76.9%	53.8%	53.8%	73.1%
r@5	84.6%	76.9%	76.9%	80.7%
r@10	84.6%	80.7%	84.6%	84.6%
MRR	0.825	0.629	0.694	0.765

- First, code slicing could remove statements that are useless for the generation of the target instance. The statements on Line 5, 6, 7 in *SourceSnippet* are removed by code slicing (as presented in code snippet 5#). These statements are redundant for the generation of the target instance *unit*.
- Second, code slicing could retrieve additional statements that are related to the generation of other instances besides the target instance. Compared to code snippet 6#, the statement on Line 1 in *SourceSnippet* is retrieved by code slicing (as presented in code snippet 5#). This statement is about the initialization of the *ASTParser* instance *astParser* which is involved in the process of retrieving the target instance *unit* from the source instance *compilationUnit*. This additional statement is necessary. Without this statement, code snippet 5# may not be successfully reused because developers might not know how to retrieve the *ASTParser* instance *astParser*.

We analyze all code snippets in *SlicedSnippets* as the aforementioned process. The results suggest that redundant statements removed by code slicing accounts for 49.6% of all statements in non-sliced *SlicedSnippets*, and additional necessary statements retrieved by code slicing accounts for 55.1% of all statements in *SlicedSnippets*.

4.6 RQ4: Influence of Ranking

As introduced in Sect. 3.6, the proposed approach ranks resulting code snippets (noted as *Ranking*) to present the top ones. The ranking contains two criteria. The first criterion is frequency (noted as $C1$) and the second criterion is Mccabe Cyclomatic Complexity (noted as $C2$). To answer RQ4, we repeat the evaluation for three times. On the first time, we disable the whole *Ranking*. On the last two times, we disable two criteria (i.e., $C1$, $C2$), respectively. Evaluation results are presented in Table 3. From Table 3, we make the following observations:

- First, disabling *Ranking* leads to significant reduction in precision and recall. When the top 1 results are inspected, the reduction is as much as 30% = (87%–60.9%)/87% for precision and 30% = (76.9%–53.8%)/76.9% for

recall. The evaluation results suggest that *Ranking* is critical for the proposed approach to achieve better performance.
- Second, disabling $C1$ has more influence on the performance of the proposed approach compared to disabling $C2$. It leads to significant reduction in precision and recall. By contrast, disabling $C2$ has little influence on the performance of the proposed approach.

4.7 Threats to Validity

A threat to the external validity is that only one framework JDT is involved in the evaluation and it may be unrepresentative. In order to support other type based instance retrievals, we prepare to involve more applications that are related to other frameworks or libraries. Another threat to the external validity is that the proposed approach is only evaluated on Java applications. Conclusions on Java applications may not hold for applications written in other languages.

A threat to internal validity is that we replace Google Code Search used in the state-of-the-art approach *PARSEWeb* with Search Code because Google Code Search is not available anymore. *PARSEWeb* may achieve better performance by using Google Code Search. Another threat to internal validity is that the manual checking of correctness for recommended code snippets could be inaccurate. Three participants are not familiar with all the recommended code snippets in the evaluation. Consequently, they may make incorrect judgements. To reduce the threat, we select three participants who all have experience in reusing JDT framework (the involved framework in the evaluation).

A threat to construct validity is the uncertainty of the existence of the solution for the requests which are selected for evaluation. We search *Stackoverflow* by keywords and manually identify requests that could be translated into type based instance retrieval queries. However, we have no idea if there really exists a code snippet that could retrieve the target instance from the source instance for a query. Maybe it is impossible to achieve that.

5 Related Work

To the best of our knowledge, *PROSPECTOR* [12] is the first approach for type based instance retrieval. The input of *PROSPECTOR* is a query in the form of (Tin, $Tout$), where Tin and $Tout$ specify the types of the source instance and target instance, respectively. The output of *PROSPECTOR* is a synthesized code snippet that instantiates a $Tout$ instance via a Tin instance. To synthesize the code snippet, *PROSPECTOR* creates a Jungloid graph according to API method signatures and adds downcast information to the graph according to a corpus of code examples. By traversing the graph from Tin to $Tout$, the approach may find out some paths that connect Tin to $Tout$. Based on such paths, it generates corresponding code snippets that retrieve the expected instance. *XSnippet* proposed by Sahavechaphan and Claypool [17] is the second approach for type based instance retrieval. The input of *XSnippet* is an object instantiation query and the output is all possible code snippets that accomplish the instantiation. To retrieve such code snippets, *XSnippet* creates a source code model for each source code file in a code example repository. After that, it employs a graph based snippet mining algorithm to mine paths that end at the instantiation specified by the given query, and statements on the paths make up code snippets. *PARSEWeb* [20] is the third and latest approach for type based instance retrieval. The input of *PARSEWeb* is a query in the form of '*Source object type -> Destination object type*'. The output of *PARSEWeb* is a set of method invocation sequences that retrieve the destination object type instance from the source object type instance. First, *PARSEWeb* searches for source code files that contain the source object type instances and destination object type instances using Google Code Search Engine. Second, it conducts a static analysis on the files to extract method invocation sequences that could retrieve the destination object from source object. All such approaches have significantly facilitated type based instance retrieval. The proposed approach differs from such approaches for the following two points. First, the proposed approach employs code slicing to include control statements into the retrieved code snippets while existing approaches fail. Second, the merging process in the proposed approach makes the resulting code snippets more complete which could handle various cases for the generation of the target instance.

Code search is applied to search for existing code snippets to reuse [6,7,21]. There exists a number of researches about code search. *Sourcerer* [1] is a search engine for open source code. It uses structural information from open source code to make fine-grained code search. *Portfolio* [14] is a code search system that retrieves and visualizes relevant functions and their usages. *CodeHow* [11] finds potential APIs related to the query by looking through descriptions and online documents of APIs to expand the query. *PRIME* [15] takes partial programs as input and outputs semantic relevant code snippets based on type state. *DeepAPI* [5] uses a neural language model called RNN Encoder-Decoder to generate API usage sequences for a given natural language query. *FACOY* [10] is a code-to-code search engine to statically find code fragments which may be semantically similar to user input code with a query alternation strategy. All such approaches significantly facilitate code search. However, they can not retrieve type based instance retrieval code snippets directly.

6 Conclusion

In this paper, we validate that type based instance retrieval code snippets are popular. We also propose an accurate and effective approach to retrieve these code snippets. The proposed approach employs code slicing to extract relevant statements (including complex control statements) that are related to the generation of the target instance. To retrieve complete code snippets, we also merge extracted code snippets whose corresponding target instances are at parallel execution paths. The proposed approach has been evaluated on twenty-six real type based instance retrieval queries. Evaluation results suggest that compared to the state-of-the-art approaches, the proposed approach improves both precision and recall.

Acknowledgments. The work is supported by the National Key Research and Development Program of China (2016YFB1000801), and the National Natural Science Foundation of China (61690205, 61772071).

References

1. Bajracharya, S., et al.: Sourcerer: a search engine for open source code supporting structure-based search. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 681–682. ACM (2006)
2. D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley Professional, Boston (2005)
3. Drori, O.: Algorithm for documents ranking: idea and simulation results. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pp. 99–102. ACM (2002)
4. Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., Cumby, C.: A search engine for finding highly relevant applications. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 475–484. ACM (2010)
5. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 631–642. ACM (2016)
6. Henninger, S.: Retrieving software objects in an example-based programming environment. In: Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 251–260. ACM (1991)
7. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: Proceedings of the 27th International Conference on Software Engineering, pp. 117–125. IEEE (2005)
8. Hucka, M., Graham, M.J.: Software search is not a science, even among scientists. arXiv preprint arXiv:1605.02265 (2016)
9. Jaskowski, W., Krawiec, K., Wieloch, B.: Multi-task code reuse in genetic programming. In: Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 2159–2164. ACM (2008)
10. Kim, K., et al.: FaCoY: a code-to-code search engine. In: Proceedings of the 40th International Conference on Software Engineering, pp. 946–957. ACM (2018)
11. Lv, F., Zhang, H., Lou, J.G., Wang, S., Zhang, D., Zhao, J.: CodeHow: effective code search based on API understanding and extended boolean model (e). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 260–270. IEEE (2015)
12. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: ACM SIGPLAN Notices, vol. 40, pp. 48–61. ACM (2005)
13. Mccabe, T.J.: A Complexity Measure. IEEE Press, Piscataway (1976)
14. McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 111–120. ACM (2011)
15. Mishne, A., Shoham, S., Yahav, E.: Typestate-based semantic code search over partial programs. In: ACM SIGPLAN Notices, vol. 47, pp. 997–1016. ACM (2012)
16. Rahman, M.M., et al.: Evaluating how developers use general-purpose web-search for code retrieval. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 465–475. ACM (2018)

17. Sahavechaphan, N., Claypool, K.: XSnippet: mining for sample code. In: ACM SIGPLAN Notices, vol. 41, pp. 413–430. ACM (2006)
18. Sim, S.E., Umarji, M., Ratanotayanon, S., Lopes, C.V.: How well do search engines support code retrieval on the web? ACM Trans. Softw. Eng. Methodol. (TOSEM) **21**(1), 4 (2011)
19. Stolee, K.T., Elbaum, S., Dobos, D.: Solving the search for source code. ACM Trans. Softw. Eng. Methodol. (TOSEM) **23**(3), 26 (2014)
20. Thummalapenta, S., Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 204–213. ACM (2007)
21. Ye, Y., Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the 24th International Conference on Software Engineering, pp. 513–523. ACM (2002)