RICE UNIVERSITY

Corpus-Driven Systems for Program Synthesis and Refactoring

by

Yanxin Lu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Swarat Chaudhuri, Chair
Associate Professor of Computer Science

Christopher Jermaine
Professor of Computer Science

Ankit B. Patel
Assistant Professor of Electrical and
Computer Engineering

Houston, Texas

April, 2019

ABSTRACT

Corpus-Driven Systems for Program Synthesis and Refactoring

by

Yanxin Lu

Software development is a difficult task. Programmers need to work with many small components in large software projects which typically contain more than thousands of lines of code. To make software development manageable, developers and researchers have deployed various programming systems and tools. These include the ones that can facilitate refactoring existing source code and even generate programs automatically. One problem with traditional program synthesis tools is that they cannot generate practical results when given large specifications due to its high complexity of the underlying problem. Furthermore, existing refactoring systems can only refactor individual components separately and fail to instantiate complete programs. To overcome these problems, we can learn useful patterns and idioms from large code corpora using machine learning techniques. Researchers have used "big code" and developed novel and practical programming tools such as Bayou [1] and JSNice [2]. In this thesis, we present two data-driven programming systems for software reuse and refactoring.

We first introduce *program splicing*, a programming methodology that aims to automate the workflow of copying, pasting, and modifying code available online. Here, the programmer starts by writing a "draft" that mixes unfinished code, natural language comments, and correctness requirements. A program synthesizer that interacts

with a large, searchable database of program snippets is used to automatically complete the draft into a program that meets the requirements. Our evaluation uses the system in a suite of everyday programming tasks and includes a comparison with a state-of-the-art competing approach as well as a user study. The results point to the broad scope and scalability of program splicing and indicate that the approach can significantly boost programmer productivity.

Next, we propose an algorithm that automates the process of *API refactoring*, where the goal is to rewrite an API call sequence into another sequence that only uses the API calls defined in the target library without modifying the functionality. We solve the problem of API refactoring by combining the techniques of API translation and API sequence synthesis. Specifically, we first translate original API calls into a set of new API calls defined in the target library. Then we use an API synthesizer to generate a complete program that uses the translated API calls. We evaluated our algorithm on a diverse set of benchmark problems, and our algorithm can refactor API sequences with high accuracy.

Although the evaluations of the techniques presented in this thesis are quite optimistic, we believe that there is room for improvement by using more sophisticated language model and advanced search algorithm for program splicing. To improve our API refactoring method, one can train statistical models by using existing API call sequence pairs. Besides these potential improvements, many problems related to "big code" still remain, and the potential of using a data-driven method to help programming is enormous.

Contents

Illustrations

Tables

Chapter 1

Introduction

Computer programming is difficult, especially when programmers need to develop components for some large software projects which typically contain more than thousands of lines of code. Hundreds or even thousands of programmers are required to collaborate and carefully work with these large software projects. In addition, these software projects tend to be extremely brittle. It is quite easy for developers to cause system-wide failure.

To make software engineering easier and manageable, some software engineers and researchers have proposed and developed programming systems and tools that facilitate software development. The most popular kind of tools is software engineering tools that help practice software engineering methodologies such as unit testing [3], project management and bug tracking [4] and source code management [5]. These software engineering tools are quite flexible and easy to use. However, if developers do not follow the underlying software engineering practices strictly, these software engineering tools would not be helpful as expected. Besides some software engineering tools, developers can also use program analysis tools [6] to discover potential bugs rigorously before deployment. These analysis tools can also monitor software executions. It allows developers to optimize software projects for better performance and sometimes to eliminate performance issues.

One type of tools that needs emphasis is synthesis tools. The goal of program synthesis is to automatically generate programs such that the generated program sat-

isfies a specification. If the given specification is comprehensive, developers can then guarantee the correctness of the generated program. One well-known synthesis tool is SKETCH [7]. SKETCH is able to generate correct programs when given comprehensive test harness. The idea of a computer being able to write programs sounds attractive, but the industry has not yet adopted these tools. One main problem is that these tools can only generate small programs; these small programs tend to serve limited purposes in the context of large software projects. Although we can feed a practical specification to some synthesis tools, these practical specifications tend to be large and it is very likely that the tool will not produce meaningful results within a reasonable amount of time. Moreover, this scaling problem is not easy to overcome. The underlying search space grows exponentially as the input size increases. Some problems are even unsolvable in theory. As a result, it is very likely that synthesis tools will not terminate in time to produce any useful results. Even though people have been trying to alleviate this problem by introducing additional hints [8] or by restricting the problem domain [9, 10, 11], those large search spaces still remain.

Another relevant type of tools is refactoring tools. Refactoring tools aim to reconstruct an existing software component without changing the functionality. Software developers need to exercise refactoring frequently to ensure that the existing software projects will not deteriorate by keeping them clean and up-to-date. Refactoring becomes even more crucial especially with the rapid evolving of the underlying software systems [12]. Researchers have deployed some tools that facilitate refactoring [13, 14, 15, 16, 17, 18, 19]. These tools can refactor software components individually, but they do not consider generating end-to-end results by combining refactored components to instantiate complete programs. One important reason is that refactoring large software projects and producing end-to-end and useful results for large

codebase is difficult. Finding a correct way to combine many small components to generate a complete and correct program requires searching in a large combinatorial space. As we have discussed previously, this cannot produce useful results within a reasonable amount of time. Similar to the problem of traditional synthesis tools, the existing software refactoring cannot scale well to handle large codebases either.

A large codebase can indeed cause many problems for traditional synthesis and refactoring tools. However, we can also treat these large corpora as data and learn useful idioms and patterns which can help mitigate the original scaling problem and solve other difficult problems related to program synthesis and refactoring. Researchers have already borrowed the technique from "big data" and have used machine learning algorithms to learn useful patterns from large code corpora. Moreover, "big code" has become popular in programming language and software engineering communities recently, since more and more open source repositories such as `Google Code`, `Github`, and `SourceForge` have made thousands of software projects and their source code available. With the help of "big code", many corpus-driven programming tools have been proposed which helped program property prediction [20, 2], API sequence prediction [21, 1, 22] and program generations [23]. Researchers have shown that using "big code" can indeed bring more capabilities to synthesis tools [23]. Practical tools such as *Bayou* [1] and `JSNice` [2] that can generate promising results have also started to appear and programmers have started to use them in practice.

Although many programming systems have been proposed, these programming tools can be categorized into two major types. The first type is based on combinatorial search. Combinatorial search plays an important role in model checking and traditional program synthesis problems [24, 25, 7, 26, 27, 28, 29, 30, 11]. The main idea is to first define a goal and the steps for reaching the goal. Programmers can

then let the computer search for a solution. Typically heuristics are defined to reduce the search space and to reduce the search time. Search-based methods have three advantages: (1) they are relatively easy to implement and they can be used to solve problems where no efficient solutions exist, (2) sometimes the algorithms can discover unexpected solutions because computers can easily discover solutions in a large search space quickly compared to humans, and (3) search-based methods can solve problems that need precision which is typically required for analyzing computer programs.

The biggest drawback of search-based methods, however, is its high algorithmic complexity. The search space grows indefinitely as the input size increases. This is the main reason most traditional model checking methods and program synthesis algorithms cannot deal with large programs [11]. Another drawback worth mentioning is that search-based methods tend to be quite fragile; they typically require precise inputs at every step, or the algorithm would not perform as expected and debugging tends to be difficult for search-based methods.

The second type is based on learning. The idea of learning is to let an algorithm improve its performance using data of past experience which helps to solving a task [31]. In our context, these data correspond to the large code corpora available on well-known repositories such as `Github` and `SourceForge`. After we feed these data to learning-based methods, they are able to capture idioms that are essential in solving the problem. Researchers started to apply learning-based methods to programming systems [20, 2, 21, 1, 22, 23], and they have shown that using "big code" can help solving the problems that were considered difficult originally and also can enable synthesis tools to produce meaningful and practical results. This is the key advantage of "big code". Examples idioms or patterns include relationships among variable names and their semantics information and API call sequence idioms. These idioms were

hard to derive without analyzing a large amount of source code. Another advantage of "big code" compared to a search-based method is its robustness; machine learning algorithms tend to use a large amount of data in which small noises are suppressed, whereas a small mistake in a search-based method can propagate along the search process and lead to big errors.

Even though data-driven programming systems are quite impactful, learning-based methods are not as accessible as search-based methods. In order to make learning-based algorithms perform well in practice, a large amount of data along with strong computing power is typically required. This also leads to a large consumption of time and computation resources which might not be available for everyone.

The programming systems we present in this thesis can also be categorized into one of those two types mentioned above. Our first programming system which automates software reuse utilizes a search-based method. Our second programming system that automates software refactoring uses a learning-based technique. Here, we assume that software reuse denotes a process of reusing existing source code by copying and pasting. For software refactoring, we consider it as a process of software rewrite without modifying the functionality. Researchers have started considering the problems of software reuse [32, 33, 34] and refactoring [13], but no systems can fully automate software reuse and refactoring. Some state-of-the-art tools [32, 13] still require humans to provide additional hints. By using a large code corpus, we claim that our works can fully automate the process of software reuse and refactoring without human intervention, and our systems can accomplish the tasks efficiently and help human developers boost their productivity during the process of software reuse.

1.1 Program reuse via splicing

We first introduce *program splicing*, a programming system that helps human developers by automating the process of software reuse. Copying and pasting from existing code is a coding practice that refuses to die out in spite of much expert disapproval [35, 36]. The approach is vilified for good reason: it is easy to write buggy programs using blind copy-and-paste. At the same time, the widespread nature of the practice indicates that programmers often have to write code that substantially overlaps with existing code and that they find it tedious to write this code from scratch.

In spite of its popularity, copying and pasting code is not always easy. To copy and paste effectively, the programmer has to identify a piece of code that is relevant to their work. After pasting this code, they have to modify it to fit the requirements of their task and the code that they have already written. Many of the bugs introduced during copying and pasting come from the low-level, manual nature of the task. In addition, programmers sometimes do not even try to fully understand the code they bring in from the internet so long as it appears to work under their specific software environment. This might pose a threat to their future software development progress, such as applying the external code to the problems which it was not targeted at initially.

Existing techniques that inspire our splicing algorithm can be divided into two areas, search-based program synthesis techniques and data-driven methods. The problem of program synthesis has been studied for decades and researchers have been applying search-based methods to tackle the problem [28, 7, 8, 29, 30, 37]. It has been well known that search-based methods can produce precise results. This is crucial when we aim to generate code snippets that need to interact with pre-written software

pieces. Examples might include matching variables that are semantically similar or equivalent.

However, search-based methods do not scale well to handle large inputs, which was typically caused by large search spaces and the complexity of the problem. This is the main reason why one of the competing system, μScalpel, is not as efficient as our splicing method. To alleviate the scalability problem, people have proved that using "big data" can be quite effective [2, 21, 38, 23, 39]. The scalability problem was greatly alleviated by using the idioms derived "big data" analyses which reduce search spaces significantly. Even though our splicing method does not use any statistical method, we still reduce our search space significantly and achieve high efficiency by relying on using natural language to search a big code corpus [40].

The primary distinction between our synthesis algorithm and existing approaches to synthesis is that we combine the ideas from search-based methods and data-driven methods by using pre-existing code. A key benefit of our approach is that it helps with the problem of *underspecification*. Because synthesis involves the *discovery* of programs, the requirements for a synthesis problem may be incomplete. This means that even if a synthesizer finds a solution that meets the requirements, this solution may, in fact, be nonsensical. This problem is especially common in traditional synthesis tools, which explore a space of candidate programs without significant human guidance. In contrast, the codelets in our approach are sourced from pre-existing code that humans wrote when solving related programming tasks. This means that our search for programs is biased towards programs that are human-readable and likely to follow common-sense constraints that humans assume.

The use of pre-existing code also has a positive effect on scalability. Without codelets, the synthesizer would have to instantiate holes in the draft with expressions

built entirely from scratch. In contrast, in program splicing, the synthesizer searches the more limited space of ways in which codelets can be "merged" with a programmer-written draft.

We present an implementation of this system, called SPLICER, for the Java programming language. SPLICER uses a corpus of over 3.5 million procedures from an open-source software repository. Our evaluation discussed in chapter 2 uses the system in a suite of everyday programming tasks and includes a comparison with a state-of-the-art competing approach [32] as well as a user study. The results point to the broad scope and scalability of program splicing and indicate that the approach can significantly boost programmer productivity.

While software reuse has been a common and important practice in software development, programmers still need to maintain their software projects after development. Otherwise, the software projects will deteriorate and be abandoned eventually, which results in a significant waste of effort and time. Therefore, we cannot neglect the importance of software maintenance. In the next section, we will begin to introduce software refactoring which plays a key role in software maintenance. We will then discuss the problem of software refactoring followed by our algorithm to automate software refactoring.

1.2 API refactoring using natural language and API synthesizer

Software refactoring typically involves reconstructing existing source code without modifying the functionality. It is important and almost a daily routine that a programmer would perform to keep their software projects clean and organized. It in-

cludes constructing better abstractions, deleting duplicate codes, and breaking down a big functionality into small pieces that are universally applicable. Maintenance is crucial because a software system can easily deteriorate and become obsolete and useless if not maintained properly and regularly. This is especially important with the rapid evolving of the underlying libraries and software systems [12]. After several decades of software development, most professional programmers have realized the importance of refactoring, and it has been used heavily and regularly in the industry. Similar to software reuse, software refactoring also inherits the drawbacks from programming. It again requires precision from programmers, and programmers tend to make mistakes when they deal with large and complex software systems. This typically involves keeping tracking of tens or even hundreds of variables and functions.

We begin to address the difficulties in software refactoring described above in this chapter. We focus on refactoring Application Programming Interface (API) call sequences. An API consists of all the definitions and usages of the resources available for external use from a software system; almost all software systems are built using APIs from other software systems. The process of API refactoring mainly consists of changing the API call sequence defined in one library into another sequence defined in another library. The benefit of performing API refactoring is identical to general software refactoring, but API refactoring has two additional benefits. First, it allows programmers to adopt obsolete source codes into the existing programming environment. Second, it can enhance the performance of existing programs by refactoring the existing program into another that uses advanced libraries and platforms which typically have better performance.

The benefit of API refactoring does not come without a few challenges. The main difficulty of API refactoring comes from discovering semantically equivalent API calls

between two libraries. It is also difficult to instantiate the new API calls using the environment's variables so that the resulting API call sequence does not alter the functionality of the original API call sequence. One of the earliest works [13] that aims to help API refactoring requires human interventions. The user of the system needs to formally specify the mapping between the API calls in two libraries, and the system only focuses on refactoring *individual* API calls instead of refactoring sequences.

Subsequent research in the area of API refactoring has been limited to the problem of API mapping or API translation. Two types of methods were developed to solve the problem of API translation. The first one involves aligning two API call sequences using a statistical model; the translations can be extracted from the alignment results [17]. This alignment method allows people to find not only one-to-one API translations but also one-to-many API translations; the downside is that it requires a large amount of API call sequences to train the underlying statistical method. The second method relies on natural language features such as Javadoc to find semantically equivalent API calls [14, 15, 41]. Since Javadoc contains descriptions on the nature of API calls, correct translations can be derived by pairing two API calls with similar Javadoc texts. Similarities can easily be derived by using a standard `Word2Vec` model [42]. The only drawback of using natural language features as the main glue is that it is difficult to discover one-to-many API translations.

To improve on the technique for API refactoring, we propose a learning-based algorithm that automates the process of API refactoring by combining the natural language technique [14] and a state-of-the-art API call sequence synthesizer called `Bayou` [1]. The input to our algorithm includes a complete program that contains an API call sequence and the name of the destination library; our algorithm can produce

another semantically equivalent program containing a new API call sequence that uses only the API calls defined in the destination library. We solve the problem in two steps. First, we translate the input API call sequences into a set of stand-alone API calls defined in the destination library using natural language features as the main driver [14, 15]. Next, we feed the stand-alone API calls into an API sequence synthesizer called *Bayou* [1] which in turn synthesizes a complete type-safe program that contains a sequence of API calls defined in the target library.

We have designed a series of benchmark problems to evaluate the accuracy of our API refactoring algorithm, and here the accuracy is defined as the percentage of corrected generated API calls. Results show that our algorithm is able to refactor API call sequences accurately, given that the two involved libraries have similar coding practices and the input sequence is not rare in the training data.

1.3 Summary

Programming is difficult, especially when hundreds of programmers are developing software projects that contain millions of lines of code. Researchers have developed synthesis tools to help software development. However, those synthesis tools cannot produce useful results in practice due to their high algorithmic complexity. With the advent of "big data" or "big code", researchers have applied "big code" to synthesis tools and developed some novel and practical tools such as Bayou [1] and JSNice [2]. In addition to these programming tools, we propose two novel programming systems for software reuse and refactoring in this thesis. We will present our data-driven methods to automate the process of software reuse and refactoring, and we begin to discuss the details of program splicing technique for reuse in chapter 2 and our algorithm for API refactoring in chapter 3.

Chapter 2

Program Splicing

2.1 Introduction

The most popular workflow nowadays consists of copying, pasting, and modifying code available online. The reason for its domination is that it is relatively easy to execute with the help of internet search. However, this process inherits the drawbacks from programming: it requires extreme precision and care from programmers similar to normal programming. When a software reuse task happens in a large and complicated software system, the cost of making mistakes and spending time on repair might exceed the benefit.

In this chapter, we present a programming methodology, called *program splicing*, that aims to offer the benefits of copy-and-paste without some of its pitfalls. Here, the programmer writes code with the assistance of a program synthesizer [29, 7] that is able to query a large, searchable database of program snippets extracted from online open-source repositories. Operationally, the inputs to synthesis include a "draft" program that is a mix of unfinished code and natural language comments, as well a correctness requirement, for example a set of test cases or a constraint on the API calls the programmer wants to invoke. The synthesizer completes the "holes" in the draft by instantiating them with code extracted from the database, such that the resulting program meets its correctness requirement. The synthesizer is invoked interactively as part of a larger program development process; initially, the draft fed

to the synthesizer may be close to empty, and the programmer is free to generate new drafts by adding code and holes to the result of a round of synthesis.

In more detail, our synthesis algorithm operates as follows. First, it identifies and retrieves from the database a small number of program snippets that are relevant to the code in the draft. These search results are viewed as pieces of knowledge relevant to the synthesis task at hand, and are used to guide the synthesis algorithm. Specifically, from each result, the algorithm extracts a set of *codelets*: expressions and statements that are conceivably related to the synthesis task. Next, it systematically enumerates over possible instantiations of holes in the draft with codelets, using heuristics to prune the space of instantiations.

We present an implementation of program splicing, called Splicer, that uses a corpus of approximately 3.5 million methods, extracted from the Sourcerer [43, 44, 45] source code repository, to perform synthesis of Java programs. Splicer uses a known method for code search to find programs relevant to a draft. The method for merging codelets with a draft is also based on existing (non-data-driven) approaches to enumerative synthesis. The key novelty of the system lies in combining these two components into an effective software engineering tool.

We evaluate our approach on a suite of Java programming tasks, including the implementation of scripts useful in everyday computing, modifications of well-known algorithms, and initial prototypes of software components such as GUIs, HTML parsers, and HTTP servers. Our evaluation includes a comparison with μScalpel [32], a state-of-the-art programming system that can "transplant" code across programs, as well as a user study with 18 participants. The evaluation shows our system to outperform μScalpel and indicates that it can significantly boost overall programmer productivity.

Now we summarize the contributions of this work:

- We propose program splicing, a methodology where programmers use a program synthesizer that can query a large database of existing code, as a more robust proxy for copying and pasting code.

- We present an implementation, called Splicer, that repurposes existing approaches to code search and synthesis and is driven by a corpus of 3.5 million Java methods.

- We present an extensive empirical evaluation of our system on a range of everyday programming tasks. The evaluation, which includes a user study, shows that our method outperforms a state-of-the-art competing approach and increases overall programmer productivity.

The rest of this chapter is organized as follows. In Section 2.2, we give an overview of our method. Section 2.3 states our synthesis problem; Section 2.4 describes the approach of program splicing; Section 2.5 presents our evaluation. Finally, we summarize this chapter with some discussion in Section 2.6.

2.2 Motivating Examples

In this section, we describe program splicing, as embodied by Splicer, using a few motivating examples.

2.2.1 Reading a Matrix from a CSV File

Consider a programmer who would like to read a matrix from a comma-separated values (CSV) file into a 2-dimensional array and then to square the matrix. However, the user does not recall in detail what API to use and how matrix multiplication is implemented exactly.

```
int[][] csvmat(String filename) {
  int[][] mat = new int[N][N];
  /* COMMENT:
  * Read a matrix from a csv file
  * REQ: String filename = ``matrix.csv'';
  * int[][] m = new int[N][N];
  * __solution__
  * return test_matrix(m); */
  ??
}
```

(a) Draft program

```
int[][] csvmat(String filename) {
  int[][] mat = new int[N][N];
  File f = new File(filename);
  Scanner scanner = new Scanner(f);
  for(int i = 0; i < r; ++i) {
    String line = scanner.nextLine();
    String[] fields = line.split(",");
    for(int j = 0; j < c; ++j)
    mat[i][j]= parseInt(fields[j]);
  }
}
```

(b) Completed draft

```
int[][] read_csv(int[][] m,int r,int c,String
     filename) {
  File f = new File(filename);
  Scanner scanner = new Scanner(f);
  for(int i = 0; i < r; ++i) {
    String line = scanner.nextLine();
    String[] fields = line.split(",");
    for(int j = 0; j < c; ++j)
    m[i][j] = parseInt(fields[j]);
  }
  return m;
}
```

(c) A database program

Figure 2.1 : Reading from CSV files

In current practice, the programmer would search the web for a program that reads from a CSV file and another one that does matrix multiplication, copy code from the search results, and modify the programs manually. In contrast, while using SPLICER, he or she writes a *draft* program in a notation inspired by the Sketch system for program synthesis [7, 46] (Figure 2.1a). This draft program declares the 2d-array `matrix`; however, in place of the code to fill this array, it has a *hole* represented by a special symbol "`??`". A hole in a program serves as a placeholder that SPLICER automatically substitutes with code, using an external snippet. In this example, the external snippet is a piece of code that reads a matrix from a CSV file.

The user is required to provide information about relevant external snippets using Javadoc style comments containing "`COMMENT`" section and "`REQ`" section above the hole or above the function. If this information is provided above the hole as showed in Figure 2.1a, then the hole itself will be replaced with the external code snippet.

The user describes the forms of external code that are relevant to the task using natural language comments. In this example, the comments contain words such as "read", "matrix" and "csv" in the "`COMMENT`" section above the hole suggesting a program that reads from CSV files. The system will use these words as a hint to search the code database. This is similar to a web search using text, but in this case it is done in a programming scenario. Finally, to ensure that the synthesized code is compatible with the code that he has already written, the programmer needs to provide some correctness requirements.

The requirements for our example are shown in the "`REQ`" section above the hole. A requirement is simply a piece of code that executes and returns true or false indicating if the solution is correct. To test the correctness of a synthesized code snippet solution, "`__solution__`" will be replaced with a candidate solution and everything under

```
int[][] csvmat(String filename) {
  int[][] mat = new int[N][N];
  ...
  int[][] mat2 = new int[N][N];

  /**
   * COMMENT:
   * square a matrix using matrix multiplication
   *
   * REQ:
   * int[][] m = {{1, 2, 3}, ...};
   * int[][] result = {{14, 20, 26}, ...};
   * __solution__
   * return test_equality(m, result);
   */
  ??

  return mat2;
}

```

Figure 2.2 : Reading CSV: draft for matrix multiplication

"`REQ`" is run in an independent environment which has no access to the variables and names in the draft program. For example, in Figure 2.1a, "`m`" is defined for storing the matrix and "`filename`" is defined for storing the filename so that the solution will have access to the filename and be able to check whether the matrix is read into "`m`". "`test_matrix`" (we omit a detailed definition of this function) is used to test the content of the matrix. The advantage of separating the testing environment from the draft program is that users can write tests in all possible ways without polluting the draft program and that is also how typical unit tests are written.

Given the draft, SPLICER issues a query to a searchable database of code snippets. The code database then returns a set of functions relevant to the current programming task, including at least one program that reads from CSV files (such an implementa-

```
1  int[][] csvmat(String filename) {
2    int[][] mat = new int[N][N];
3    File f = new File(filename);
4    Scanner scanner = new Scanner(f);
5
6    for(int i = 0; i < N; ++i) {
7      String line = scanner.nextLine();
8      String[] fields = line.split(",");
9      for(int j = 0; j < N; ++j)
10     mat[i][j] = Integer.parseInt(fields[j]);
11   }
12
13   int[][] mat2 = new int[N][N];
14   for(int i = 0; i < N; ++i) {
15     for(int j = 0, s = 0; j < N; ++j) {
16       for(int k = 0; k < N; ++k) {
17         s += mat[i][k]*mat[k][j];
18       }
19       mat2[i][j] = s;
20     }
21   }
22   return mat2;
23 }
24
```

Figure 2.3 : Reading CSV: Complete Program

tion is shown in Figure 2.1c). The system now extracts a set of *codelets* — expressions and statements — from these functions, and uses a composition of these codelets to fill in the hole in the draft. The completed draft is showed in Figure 2.1b.

After getting the code that reads a matrix from a csv file, the user now focuses on the second part of the task, which is matrix squaring using matrix multiplication. The previous code is now extended into a new draft, which has a hole for the matrix multiplication code, some comments and requirements. This draft is shown in Figure 2.2. SPLICER now searches the code database for snippets that perform ma-

trix squaring using normal matrix multiplication and merges these snippets into the existing code, while ensuring that all requirements are met. The complete program resulting from this process is shown in Figure 2.3.

As shown in the example, SPLICER can be used in an iterative and interactive manner. A programmer can start writing code as usual, and then bring in external resources from the web into the existing codebase as needed. In this respect our approach is similar to copying and pasting code. The difference is that SPLICER automates the process of finding and modifying relevant code, and guarantees a certain level of reliability by ensuring that the output program meets all its requirements.

2.2.2 Face Detection using OpenCV

In previous examples, we relied on input-output tests to verify the correctness of a solution. Now we consider the use of program splicing in the implementation of *face detection*, a computer vision task in which input-output tests are hard to specify, requiring the use of an alternative form for correctness requirement. Specifically, the requirements that we use are constraints on sequences of API calls that a program makes, given in the form of a finite automaton.

Figure 2.5 shows a draft program for this task. In this example, a user wants to use a `CascadeClassifier` object from OpenCV to detect faces from an input image called `lena.jpg`. The output image named `faceDetection.png` should have the same picture with a rectangle drawn above the faces.

The API call constraint for the task is shown in Figure 2.4. This requirement describes a sequence of object creation and API invocation actions performed during face detection. To check the API call requirement, SPLICER runs the candidate solutions under an environment where necessary functions and variables are defined to

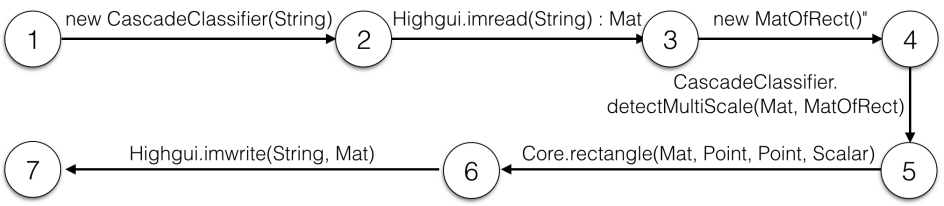


Figure 2.4 : API call sequence constraint for face detection

keep track of the program state which is checked against the requirement. For example in Figure 2.5, SPLICER loads the requirement (line 3), runs a candidate solution and checks internally (line 5) if the solution has created a face detector (`_has_detector_`), has loaded an image (`_has_image_`) and etc., and it ensures things are completed in order. Predicates like "`_has_detector_`" are defined along with the API constraints. In Figure 5 for example, they are defined inside `FaceDetectionTest.java`.

```
1     /**
2      * COMMENT:
3      * Doing face detection using OpenCV
4      *
5      * REQ:
6      * API_cons("FaceDetectionTest.java");
7      * __solution__
8      * run_and_test(_has_detector_ && _has_image_ &&
9      *              _has_detection_ && _image_written_);
10     */
11 public void run() {
12   String input_img = "lena.jpg";
13   String output_img = "faceDetection.png";
14   CascadeClassifier detector = new CascadeClassifier(??);
15   ??
16 }
17
```

Figure 2.5 : Face Detection: Draft Program

```
1  public void run() {
2    String input_image = "lena.png";
3    String filename = "faceDetection.png";
4    CascadeClassifier detector =
5      new CascadeClassifier(getClass().getResource("lbpcascade_frontalface.xml").getPath());
6    Mat image = Highgui.imread(getClass().getResource(input_image).getPath());
7    MatOfRect faceDetections = new MatOfRect();
8    detector.detectMultiScale(image, faceDetections);
9    for(Rect rect : faceDetections.toArray()) {
10     Core.rectangle(image, new Point(rect.x, rect.y), new Point(rect.x + rect.width, rect.y + rect
       .height), new Scalar(0, 255, 0));
11   }
12   Highgui.imwrite(filename, image);
13   }
14
```

Figure 2.6 : Face Detection: Complete Program

While the requirement is more low-level than unit tests, we note that it frees users from specifying small details such as what configuration file to be used, the color for drawing rectangles on faces and the order of specifying the four corners of rectangles. SPLICER uses this requirement to filter out many of the candidate programs that it considers during synthesis. Only a few solutions satisfy the requirement, and the user could easily pick the correct one shown in Figure 2.6.

Note that the external code information in this example is provided above the function, as shown in Figure 2.5. In this case, SPLICER will replace all the holes inside this function with possibly different external code snippets in a single run using a single set of relevant programs.

2.3 Problem formulation

In this section, we define the problem of program splicing.

Language Definition As mentioned earlier, a draft program in our setting consists of incomplete code and a set of natural language comments. We start by specifying the language of code permitted in our drafts.

Our approach accepts code in a subset $\mathcal{L}$ of Java, abstractly represented by the following grammar. In summary, the grammar permits standard imperative expressions and statements over base and array types, as well as a symbol `??` representing holes.

$$\langle expr \rangle ::= id \mid c \mid \langle expr \rangle \; binop \; \langle expr \rangle \mid unaryop \; \langle expr \rangle$$
$$\mid \quad \mathrm{f}(\langle expr \rangle, \ldots, \langle expr \rangle) \mid id := \langle expr \rangle \mid ??$$
$$\langle stmt \rangle ::= \textbf{let } id = \langle expr \rangle \mid \textbf{if } \langle expr \rangle \; \langle stmt \rangle \; \langle stmt \rangle$$
$$\mid \quad \textbf{while } \langle expr \rangle \; \langle stmt \rangle \mid \langle stmt \rangle \; ; \langle stmt \rangle \mid ??$$
$$\langle program \rangle ::= id \; (\langle expr \rangle, \ldots, \langle expr \rangle) \; \langle stmt \rangle$$

In this grammar, c represents a constant, id represents an identifier, f represents external functions (API calls), and *binop* and *unaryop* respectively represent binary and unary operators. We assume that a standard type system is used to assign types to expressions and statements in this grammar. The actual language handled by our implementation goes somewhat beyond this grammar, permitting arrays, objects, data structure definitions, a limited form of recursion, and syntactic sugar such as for-loops.

The special symbol `??` in the grammar represents two kinds of holes. *Expression holes* is a placeholder for a missing expression. A *statement hole* is a placeholder for a missing statement.

The semantics of a program with holes can be defined as a set of complete (hole-free) programs obtained by instantiating the holes with expressions and statements.

The semantics of a complete program is defined in the standard way. We skip the formal definitions of these semantics for brevity.

Requirement Aside from a draft, an input to a program splicing problem includes a *requirement*. This requirement is not expected to be a full correctness specification. Specifically, our implementation permits two classes of requirements: input-output tests, and finite automata that constrain the sequences of API calls that a program can make. We assume a procedure to conservatively check whether a given complete program satisfies a given set of requirements. For requirements that are input-output tests, this procedure simply evaluates the program on the tests. The procedure for automaton constraints is based on a standard, sound program analysis.

Program Splicing Let $P_s \in \mathcal{L}$ be a draft program with one or more holes. Let $DB \subseteq \mathcal{L}$ be a database containing programs with *no* holes. Our objective is to use the programs from DB to complete holes in P_s. Specifically, we use the expressions (similarly, statements) from DB to complete the expression holes (similarly, statement holes) in P_s. Naturally, such an instantiation of the holes can be performed in many ways. Our goal is to do this instantiation such that the resulting program passes the requirement.

More precisely, consider the set $\mathcal{C}$ of all *codelets* — subexpressions and statements — that appear in programs from DB. Let $\mathcal{P}$ be the set of complete programs obtained by instantiating the holes of P_s by appropriately typed codelets in $\mathcal{C}$. Let $U : \mathcal{L} \to \{\mathit{True}, \mathit{False}\}$ be a function that maps a complete program in $\mathcal{L}$ to a boolean value indicating whether the input program passes the requirement accompanying P_s. The splicing problem is to find a program $P_c^* \in \mathcal{P}$ such that $U(P_c^*) = \mathit{True}$.

2.4 Method

In this section, we present a specific solution to the splicing problem, implemented in the SPLICER system. Our synthesis problem has two key subproblems: *code search* and *hole substitution*.

Code Search Given a program $P_s \in \mathcal{L}$, search a large corpus containing thousands of programs for a set of relevant programs such that the retrieved programs contain the codelets that are needed to fill holes. The desired property of the code search technique should be that the retrieved programs should contain the exact codelet we need within a short period of time.

Hole Substitution Given multiple database programs S_d, we would like to search for the correct codelets to fill the hole. Multiple programs combined consist of a large number of codelets. The key challenge here is to prune the search space such that we can efficiently get the exact codelet we need and ensure the necessary codelets will not be dropped.

2.4.1 Searching for programs

In this section, we describe the code search techniques from Kashyap et al. [40] with a modification employed to query a large database of programs effectively. This is the first step in our workflow: to find candidate functionality from the program database to complete the draft program. Given the word hints below "`COMMENT`" in the Javadoc and also the variable names in the draft program, SPLICER does a code search and returns a set of relevant programs.

An important goal of the code search component is to have a quick response when searching large amounts of code. To accomplish this, various code features are extracted from a large corpus of open source code. These code features—along with the

corresponding source code—are stored in a program database. The program database is a scalable object-store database that allows for fast similarity-based queries.

A query issued to the program database includes code features extracted from the draft program, along with associated weights indicating the relative importance of the code features. The program database computes the k nearest neighboring corpus elements to the query, using the code features stored, associated weights, and similarity metrics defined on each code feature. The result of the query is presented as a ranked list of source code corresponding to the k-nearest neighbors.

Below we describe the features extracted and the associated similarity metrics.

Natural language terms. For this feature, we extract the function name, comments, local variable names, and parameter names of a function. Such extracted natural language (NL) terms are then subjected to a series of standard NL pre-processing steps, such as splitting words with underscores or camel-case, removing stop words (including typical English stop words, and those specialized for Java code), stemming, lemmatization, and removing single character strings. Additionally, we use a greedy algorithm [47] for splitting terms into multiple words, based on dictionary lookup. This is to handle the case where programmers combine multiple words, without separating the words with underscores or camel-case, when naming functions and variables.

After NL pre-processing, we compute a tf-idf (term frequency-inverse document frequency) score for each NL term. Each function is considered as a document, and the tf-idf is computed per project. We give the function name term an inflated score (5× more than other terms) because it often provides significant information about a function's purpose. The similarity between two functions is measured by taking the cosine-similarity of their NL terms, together with their tf-idf values. Below is an

example of NL terms features for the draft showed in figure 2.1a.

```
"read":0.10976425998969035, "matrix":0.658585559938142,
"csv":0.10976425998969035, ...
```

Names. Here, we extract all the variable names, the name of the function, and perform some basic normalization such as splitting camel case and underscores. The similarity metric used is the Jaccard index on sets of names.

The code search method is described extensively in [40] and the main difference is that our similarity search is primarily driven by the natural language term features, with variable names and function names providing additional context around the hole in the query code. We give more weights to natural language term features and less weights to variable names and function names. The reason is that the most important hint in the draft code is the comment, because users are required to describe the code they want to synthesize. However, variable names and function names must not be treated as equally important, because sometimes variable names and function names might be totally irrelevant to the code they want to synthesize. For example, users might leave comments saying that they want the code that reads a matrix from a csv file, but it is totally possible that the surrounding context is all about matrix calculation.

2.4.2 Program completion

After we have retrieved a set of programs from the program database, our next step is to complete the draft by synthesizing codelets. A codelet here is a sequence of program statements or a set of expressions from the programs retrieved from the database during code search. Note that unlike other traditional synthesis techniques [28, 29,

37, 48, 49, 50, 51, 30], our synthesis method does not generate code from scratch, but instead it uses codelets from a large code corpus. For each database program paired with the given partial program, we spawn a thread to do the code completion task, parallelizing the process. A code completion task consists of the following steps:

Hole substitution

The first step is to use the codelets from the retrieved program to substitute the holes in the draft. Procedure 1 shows the algorithm. We start by checking whether there is any hole in the draft at line 1. If not, we move on to the merging step. Otherwise, we start injecting codelets into the draft. For each hole, we iterate all the codelets starting from the smallest one and check whether the injection is valid using our heuristics at line 3. If so, we then substitute the hole with the codelet at line 4 and then continue injecting more codelets by recursively calling itself at line 5 until we finish filling all the holes. When no more holes exist in the draft program, we then merge the codelets into the existing codebase, which is explained in detail in later section. If at some point injecting a codelet is not successful, we backtrack and try another codelet. Next, we discuss our heuristics used in the step of hole substitution.

Synthesizing expressions If we are searching for a substitution n for an expression hole h, we ensure n and h are of the same type. In addition, we can also consider the *roles* of h and n. The intuition is that we only consider the codelet that serves as the same role by looking at the parent of n and the parent of h in the parse tree. If the parents of n and h are not of the same kind, then we discard n and look for another codelet. Figure 2.7 illustrates the idea. If we are looking for a codelet to replace a hole representing the `rval` inside an assignment statement, our target codelets are more likely to be the `rval` of other assignment statements. We can then

Procedure 1 fill

Require: A draft program, $P_s \in \mathcal{L}$ and a database program $P_d \in \mathcal{L}$

Ensure: A complete program P_c

```
1: if not has_hole(P_s) then return merge(P_s) end if
2: for h ← next_hole(P_s), n ← next_codelet(P_d) do
3:     if valid(P_s, h, n) then
4:         P'_s ← substitute(P_s, h, n)
5:         P_c ← fill(P'_s, P_d)
6:         if P_c ≠ null then return P_c end if
7:     end if
8: end for
9: return null
```

just consider those codelets as substitutions and ignore other codelets. The same can be applied if we want to synthesize the code for the guard of a condition, for example.

Synthesizing statements When we are searching for substitutions for a statement hole h, we need to consider a sequence of statements from the database program. We define a sliding window of various lengths and use that to scan the database program to identify the statement sequence we would like to use to substitute for h. We also scan the sequences under loops and conditions. We then use each codelet to fill the hole.

Code merging

One problem with using the codelets from the database programs is that the naming schemes are different from the ones in the original draft program. Therefore, after

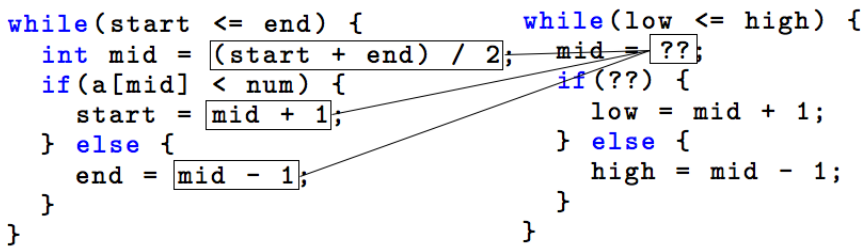


Figure 2.7 : Matching for expression codelet

we have completed the draft program, we search for reference substitution such that the resulting program refers back to the data defined in the draft program, which is quite similar to code transplantation [32].

The algorithm is shown in Procedure 2. The task here is essentially searching for a mapping between the references across two programs. We first check whether we have undefined references in the program at line 1. If not, we check the program correctness against the requirement at line 2. If it is correct, then we have a solution. If there is still undefined reference in the program, we then try to rename each undefined reference u to another defined reference r at line 7. We repeat by recursively calling itself until no more undefined names exist in the program. We guide the search by using types. When we are considering renaming u to r, we rename u only if their types are the same. If at any point the algorithm cannot rename a reference due to the lack of available target references in another program, the algorithm will backtrack and try another renaming for a previous reference. This reference substitution step is performed every time we complete a draft and thus the whole algorithm suffers from exponential blowup. To ensure the algorithm terminates, we set a time limit on the entire search process.

Procedure 2 merge

Require: A completed draft program, $P_s \in \mathcal{L}$ and a database program $P_d \in \mathcal{L}$

Ensure: A correct completion P_c

if no_undefined_refs(P_s) **then**
    if is_correct(P_s) **then return** P_s **end if**
    return *null*
end if
for $u \leftarrow$ next_undefined_ref(U), $r \leftarrow$ next_ref(P_d) **do**
    if same_type(P_s, u, r) **then**
        $P'_s \leftarrow$ substitute(P_s, u, r)
        $P_c \leftarrow$ merge(P'_s, P_d)
        if $P_c \neq null$ **then return** P_c **end if**
    end if
end for
return *null*

Testing

After we have finished renaming all references in a completed program, we validate the solution against the requirement either in form of a predefined input-output test suite or a predefined API call sequence constraint given as a finite automaton. If users provide IO tests, we run the solution on the provided test suite to validate its correctness. If an API call sequence constraint is given instead, we encode the constraint into Java source code in which API calls are captured and new variables are defined to keep track of the current state in the finite automaton. When the complete program is run, the constraint will be automatically checked and thus the correctness is determined. We also set a time limit for program execution to ensure termination. Notice that we could let the synthesis algorithm produce multiple solutions by letting it continue the search after a correct completion is found. If there are multiple correct completions, we will rank them in the order they appear and return as many solutions as required. On the other hand, we can also easily add a selection function to choose the best solution.

2.5 Evaluation

Our goal is to evaluate the performance of Splicer and its ability to complete a draft program. The experiment consists of completing a set of draft programs given a code database where a set of relevant statistics for each run is recorded. In addition, we show the results of a user study where we test whether our synthesis tool could increase programming productivity.

2.5.1 Benchmarks

In this section, we briefly describe our benchmark problems followed by the experiments and the results. We evaluate the performance of SPLICER and select a set of benchmark problems with corresponding draft programs to automate the process where users try to bring external resources from the web and merge them into the existing codebase.

It is desirable to compare SPLICER with existing synthesis methods including Sketch [7], syntax-guided synthesis [52], code reuse tools [53, 54, 55, 56] or other statistical methods [34, 57]. However, none of these methods are comparable, because (1) traditional synthesis methods do not search for or use existing source code, (2) code reuse methods do not consider programs at the granularity of statements and expressions and (3) some methods such as SWIM [34] and anyCode [57] only aim to synthesize API-specific code snippets. Specifically, we fed a standard binary search draft program with a few expression holes to Sketch and it was not able to complete the draft within 30 minutes. In contrast, our splicing system could generate the correct expressions within 5 seconds after the code search is complete. Moreover, our splicing system could generate code snippets while Sketch cannot handle statement synthesis problems.

Code transplantation or μScalpel [32] is the most similar system to our work and we will use μScalpel for comparison with the correct donor programs provided to μScalpel. Notice that we cannot apply μScalpel to some of our system-related benchmark problems, because μScalpel targets at C programs instead of Java. Since systems programs in C and in Java tend to be very different in terms of number of variables, types and system calls which makes the comparison unfair. Therefore, we only compare our tool with μScalpel on some benchmark problems where the

differences in the solutions are not significant.

Our benchmark problems consist of synthesizing components from online repositories and we include 15 benchmark problems. These benchmark problems were chosen were chosen to reflect a diverse set of everyday programming tasks. Accordingly, they comprise textbook programming tasks, Stackoverflow questions, and tutorials of tools like OpenCV. The tasks also meet three criteria: (a) the problems should come from a diversity of domains, (b) the tasks should represent common programming problems, and (c) there should be adequate number of programs relevant to the problems in the code corpus. The draft program for most benchmark problem contains one or two statement and expression holes. Each draft program has its own comments and correctness requirements. Most benchmark problems use typical input-output tests except for "Echo Server", "Face Detection" and "Hello World GUI" where API call sequence constraints are used to check the correctness. Here, we highlight two draft programs from the benchmark problems.

LCS Table Building A user calculates the longest common subsequence of two integer arrays, and she has written a draft program with the code snippets to extract the subsequence from the table and display the result. A hole is left for the code that builds the table for running dynamic programming algorithm.

HTTP Server A user would like to set up an HTTP server that serves the content of a text file. She wrote a draft program which has a HTTP request handler, but she does not remember how to read from a text file and how to set up an HTTP server. Two holes are left for the code that reads from a text file and the code that sets up an HTTP server. In addition, she also leaves a hole for the response status code in the request handler. Figure 2.8 shows the draft program.

```
1   /*
2   * COMMENT: Setting up an HTTP server that serves the
3   *          content of a local file
4   * REQ:
5   * import com.sun.net.httpserver.*,
6   * import java.io.OutputStream;
7   * __solution__
8   * HttpServer server = http("http_test.txt", 23456);
9   * test_server(new URL("http://localhost:23456/"));
10  * server.stop(0);
11  */
12  public HttpServer http(String filename, int port) {
13    String content;
14    // read the content of the file
15    ??
16    HttpServer server;
17    HttpHandler handler = new HttpHandler() {
18      public void handle(HttpExchange he) {
19        he.sendResponseHeaders(??, content.length());
20        OutputStream os = he.getResponseBody();
21        os.write(content.getBytes()); os.close();
22    }};
23    // set up an http server
24    ??
25    return server;
26  }
27
```

Figure 2.8 : Setting up an HTTP server that serves the content of a text file

2.5.2 Experiments

We implemented program splicing in Scala 2.12.1 based on 64-bit OpenJDK 8 and we used BeanShell [58] and Nailgun [59] to test all the completed draft programs. For each benchmark problem, we ran SPLICER on the draft program we derived. These experiments were conducted on a 2.2GHz Intel Xeon CPU with 12 cores and 64GB RAM. For each program, we record the runtime for synthesis and we stop the synthesis once the time exceeds five minutes. To roughly have a sense of the search space size, we list the number of variables and holes in each draft program, the line number and the number of database programs we use for synthesis. Finally, we list the LOC of the draft program and its completed version. Our corpus comes from the `Maven 2012` dataset from Sorcerer [43, 44, 45]. We extracted over 3.5 million methods with features from this corpus.

Synthesis Algorithm Evaluation

Table 2.1 shows the results for each benchmark problem with $k = 5$ where k is the number of database programs we retrieve. We set $k = 5$ because empirically five programs are usually sufficient to ensure that the retrieved programs contain the target codelet we want to synthesize. In addition, we put more weight on features that consider comments and variable names in the k-nearest-neighbor search. The choice on weight selection is explained in section **??**.

According the results showed in Table 2.1, data-driven synthesis works for all benchmark problems. The time required for most code searches which is based on k-nearest-neighbor search is approximately 15 seconds, meaning that the code search is very efficient, given that we have millions of functions in the database. For most of the benchmark problems, our method was able to complete the draft program in

Benchmarks	Synthesis Time	No Roles	No Types	LOC	Var	Holes (expr-stmt)	Test	μScalpel
Echo Server	3.0	4.0	17.1	9-17	1	1-1	C	N/A
Sieve Prime	4.6	33.0	8.8	12-17	2	2-1	3	162.1
Collision Detection	4.2	6.3	5.3	10-15	2	2-1	4	N/A
Collecting Files	3.0	6.0	27.0	13-25	2	1-1	2	timeout
Face Detection	8.1	12.2	43.1	21-28	2	1-1	C	N/A
Binary Search	15.4	16.0	47.9	12-20	5	1-1	3	timeout
Hello World GUI	16.0	timeout	timeout	24-33	4	1-2	C	N/A
HTTP Server	41.1	87.4	timeout	24-45	6	1-2	2	N/A
Prim's Distance Update	61.1	66.4	timeout	53-58	11	1-1	4	timeout
Quick Sort	77.2	191.5	217.6	11-18	6	1-1	1	timeout
CSV	88.4	timeout	timeout	13-23	4	1-2	2	timeout
Matrix Multiplication	108.9	151.9	timeout	13-15	8	1-1	1	timeout
Floyd Warshall	110.4	timeout	timeout	9-12	7	1-1	7	timeout
HTML Parsing	140.4	timeout	timeout	20-34	5	1-2	2	N/A
LCS	161.5	168.8	timeout	29-36	10	0-1	1	timeout

Table 2.1 : Benchmarks. "C" in the "Test" column indicates an API call sequence constraint is used to check the correctness

under two minutes and the number of tests required is no more than five, indicating that users of SPLICER do not have the burden of writing too many tests. In general, a set of tests is considered sufficient if a complete code coverage is achieved in the desired target program. Notice that for "Echo Server", "Face Detection" and "Hello World GUI", a letter "C" is used to signal an API call sequence constraint being used to test the correctness. We can also see that synthesis takes more time as the number of holes and the number of variables increase. Having more holes, more variables and sometimes more lines leads to larger combinatorial search space for hole substitutions with codelets, and more variables increase the search space for code merging and renaming.

Impact of type matching and role matching Types ensure the solution will type check. In addition, role matching eliminates the expression substitutions where the role of a candidate expression is different from the role of a hole. To understand their impact, we record the synthesis time without using types, which is showed in the "No Types" column of Table 2.1. The "No Roles" column shows the runtime without role matching. We can see that using types and roles can reduce a large amount of search space, although types seem to be more effective. These heuristics become more and more important for larger draft programs as the number of variables increases. Without types and role matching, our synthesis algorithm timed out for some harder benchmark problems. Notice that role matching is applied when we synthesize expressions, as we cannot apply role matching when synthesizing statement sequences, and thus we do not see any difference in the "LCS" benchmark problem.

μScalpel Comparison Code transplantation [32] is very similar to our work, except that it does not consider using a large code corpus. However, it is still worthwhile to conduct a series of performance comparisons, since μScalpel also extracts

```
1   /**
2    * TODO 1: Use Sieve of Eratosthenes to test primality
3    *          of the given integer.
4    */
5    static boolean sieve(int n) {
6      boolean[] primes = new boolean[100];
7      return primes[n];
8    }
9
10  /**
11   * TODO 2: Test the sieve of Eratosthenes you've just
12   *  written. Make sure to test the program with the
13   *  following inputs: n = {1, 2, 3, ..., 73}
14   *  Return true if the program is correct.
15   */
16   static public boolean test() {
17     return false;
18   }
19
```

Figure 2.9 : `Sieve of Eratosthenes` Skeleton with Tests

code snippets from external programs, or *donor* programs. We ran μScalpel on some of our benchmark problems with correct donors specified. Notice that μScalpel has an advantage over SPLICER under this setting, because μScalpel does not need to search for relevant programs from a code corpus. Nevertheless, even with such an advantage, most of the runs could not finish within five minutes except for "Sieve Prime" which is relatively easy. Even though we did not run μScalpel on all benchmark problems, it is reasonable to believe that the performance of μScalpel (based on genetic programming) is not as efficient as SPLICER, which is based on enumerative search.

User Study

We performed a user study to evaluate the extent to which SPLICER can help human developers. Now we describe this study.

Study setup. We recruited 12 graduate students and six professional programmers and developed four programming problems (described later in this section). Each participant was asked to complete all four programming problems using a web-based programming environment. Per person, two problems were completed using program splicing (we subsequently call this a "with" task), and two without (a "without" task). "With" and "without" tasks were assigned to participants randomly.

To simulate an industrial programming setting where an engineer is asked to develop a code meeting a provided specification, for each task, participants were given a description of the program they need to implement, and a description of the test cases they need to write to verify the correctness of the program. Figure 2.9 shows an example skeleton program for the "without" task on the Sieve of Eratosthenes programming problem. For the "with" task, the draft program is almost identical to the `sieve` function in Figure 2.9 except that there is a hole after the array declaration and participants need to put in comments and requirements.

When completing both "with" and "without" tasks, participants were encouraged to use relevant code snippets from the Internet. For the "with" tasks, participants were asked to use SPLICER to provide at least one candidate solution to the programming problem, but then they could choose to use that candidate, or not use it. Before using the web-based programming environment and SPLICER, they were asked to finish a warm-up problem to be familiar to the programming environment and SPLICER to eliminate learning effects.

To evaluate whether SPLICER could boost programming productivity, we recorded the durations the participants used to correctly complete each problem. To determine if there is a statistically significant difference in completion time for "with" versus "without" tasks for the same programming problem, we define the following null hypothesis:

H_0^P = "For programming problem P, the expected 'without' task completion time is no greater than the expected 'with' task completion time."

If this hypothesis is rejected at a sufficiently small p-value for a specific programming problem, it means that it is likely that the average completion time is smaller for the "with" task than the "without" task, and hence program splicing likely has some benefit on the problem. Given the times recorded over each problem and each task, we use bootstrap[60] to calculate the p-value for each problem. The bootstrap works by simulating a large number of data sets from the original data by re-sampling with replacement, and the p-value is approximated by the fraction of the time when the null hypothesis holds in the simulated data sets. In addition to measuring time, we also recorded the number of times that "with" task participants for each problem asked the program splicing system for help. Typically the participants would stop using our system after they have received a useful codelet, and so many requests may indicate an inability of the system to produce a useful result.

Programming Problems. Now we describe the four programming problems used in the study.

`Sieve of Eratosthenes`: Implement the Sieve of Eratosthenes to test the primality of an integer. This is an interesting problem because it is purely algorithmic, and further, codes to solve this problem are ubiquitous on the Internet. We expected

Splicer to be of little use, because an Internet search should result in many different Sieve programs which should be trivial to tailor to the problem. Given this and the fact that test codes are so easy to write, we expected participants will use the least amount of time to finish this problem, regardless of whether they are given a "with" or "without" task.

`File Name Collection`: Collect all file names under a directory tree recursively and return the list of file names. We chose this problem because it represents an easy systems programming problem. Further, there is no standard solution to this problem, while it is still quite easy to write tests. Therefore we expected Internet search to be less useful, whereas program splicing might be quite helpful.

`CSV Matrix Multiplication`: Read a matrix from a CSV file, square the matrix and return it as a 2d-array. This problem includes a combination of system programming and algorithmic programming. We chose this problem expecting that "with" task programmers would need to use Splicer multiple times in an interactive manner to generate two independent code snippets. Given this, we expected that the time gap between the "with" task and "without" task participants to be smaller.

`HTML Parsing`: Read and parse an HTML document from a text file, store all links that contain a given word into a result list and return the result list. This is the most difficult problem among the four. Not only would those "with" task participants need to use Splicer multiple times, but they are required to write tests for HTML manipulation since program splicing necessitates that participants manually provide HTML to build test cases that are used to validate the correctness of the code for extracting links from the parsed HTML document. At the same time, the JSoup [61] HTML parsing library that we asked participants to use has rather comprehensive and straight-forward documentation. Hence, we expected that time gap between "with"

and "without" task participants would be the smallest among the four problems.

Results. Figure 2.11 shows the p-values for each programming problem, as well as the number of times code splicing was invoked for each problem's "with" task. Figure 2.10 shows time spent on each submission with and without splicing, including the average time, as box plots. We can see that for most programming problems except for `HTML`, the average time used to finish the "with" task is is significantly lower than the time required to finish the "without" task. The p-values in Figure 2.11 are also small enough for us to reject the null hypotheses (stating that there is no utility to program splicing) with over 99% confidence. Note that the average number of program splicing invocations for most problems (except `HTML Parsing`) is very close to one, meaning that program splicing could return codelets that the participants could use to complete the problem with only one try. We argue that this also indicates that Splicer is rather easy to use, and is indeed able to boost programming productivity in many cases. As the level of difficulty of the problem increases, so does the benefit of using Splicer.

It is, however, useful to consider the `HTML Parsing` programming problem, which is the one case where program splicing was not useful. Why is this? After careful investigation, we believe that there are two reasons program splicing did not help. First, the documentation of the HTML parsing library used, `JSoup` [61], is very comprehensive and well-done. Hence the problem was easy. Second, it is very easy to make mistakes when writing tests, which require developing correct HTML code and inserting it in a test. We found that participants typically forgot to escape quote characters within a string when loading a variable containing even very simple HTML. The difficulty in writing tests meant that program splicing was less helpful. That said, writing tests has independent value, and if the difficulty in writing tests was the key

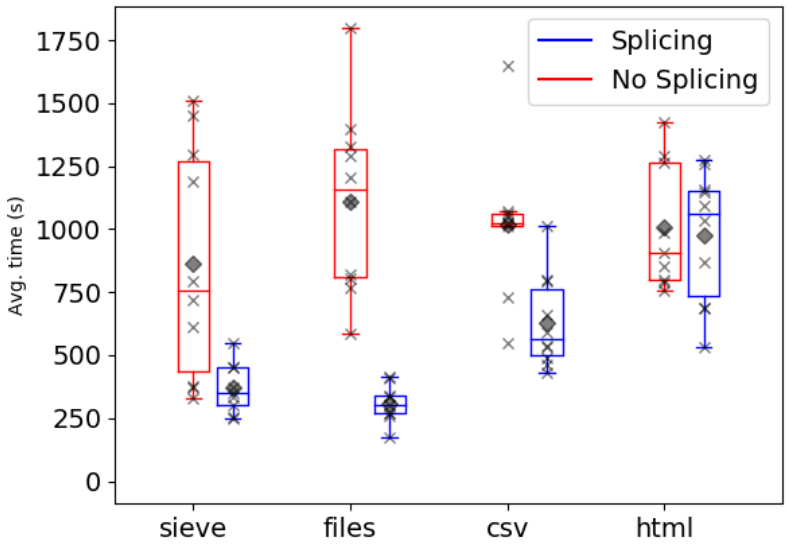


Figure 2.10 : Time spent on each programming problem, with and without splicing.

impediment to using splicing, it may not be a strong argument against the tool.

We close this subsection by asking: When is program splicing likely most useful for programmers? One surprising case seems to be programming problems that are deceptively simple, containing intricate algorithmics (loops and recursion) that programmers tend to have a difficult time with. `Sieve of Eratosthenes` falls in this category. The Sieve appears to be very simple, and so we initially expected splicing to be of little use. However, due to the perceived simplicity, we found that "without" participants tended to write their own solutions without consulting the Internet (even though we encouraged Internet use)—and this over-confidence resulted in buggy programs and longer development times. Use of program splicing protected "with" participants from such difficulties.

We also found splicing to be useful when documentation is lacking and there is not a standard way of doing things. Consider `CSV` and `Collecting File Names` where the official Java documentation does not provide any code snippets showing how to

Problem	p-value	Avg. Number of Invocations
Sieve	0.00008	1.2
CSV	0.0002	1.2
Files	0	1
HTML	0.5	2.45

Figure 2.11 : p-value at which the null hypothesis is rejected, and the average number of program splicing invocations for each programming problem.

parse a CSV files or how to collect file names under a directory subtree. "Without" participants had to rely on combing through solutions from StackOverflow [62], where multiple solutions exist, using different libraries, each with various pros and cons. Program splicing cuts out the need for manual searching and understanding many different possible solutions—if the splicing succeeds and passes the provided test cases, the user can be relatively confident that the provided solution is correct.

Limitations

Finally, we summarize the limitations of our evaluation:

1. Our corpus contains over 3.5 million methods with features which ensured that we always found relevant programs during our experiments. However, for more obscure programming problems, it may not always be possible to find helpful code in such a database.

2. Like most synthesis algorithms, the time complexity of our algorithm is exponential, and so it is limited in its ability to handle very large draft programs.

3. It is unclear how to ensure that a user-provided test suite is comprehensive enough to ensure the correctness of the synthesized program.

2.6 Summary

In this chapter, we introduce *program splicing*, a synthesis-based approach to programming that can serve as a principled and automated substitute for copying and pasting code from the Internet. The main technology is a program synthesizer that can query a database containing a large number of code snippets mined from open-source repositories. Our experiments show that it is possible to synthesize missing code by combining such database queries with a combinatorial exploration of a space of expressions and statements. We also conducted a user study and the results show that our method could indeed boost programming productivity.

Chapter 3

API Refactoring

3.1 Introduction

Software refactoring involves reconstructing existing source code without modifying the functionality. Refactoring is almost a daily routine to keep software projects clean and organized. It includes modifications such as constructing better abstractions, deleting duplicate codes, and breaking down a big functionality into small pieces that are universally applicable. Keeping source code clean and organized is crucial because software projects deteriorate. This deterioration increases the cost of maintaining the code and even prevents programmers from implementing new functionalities or debugging, especially with the rapid evolving of the underlying libraries and software systems [12].

In this chapter, we focus on refactoring Application Programming Interface (API) call sequences. An API consists of all the definitions and usages of the resources available for external use from a software system. Almost all software systems are built using various APIs from other software systems. The goal of performing API refactoring is similar to software refactoring: rewrite an API call sequence into another sequence without modifying the functionality. The difference is that the new API call sequence will use a new library or will be targeting a new platform. Moreover, the benefit of performing API refactoring is identical to software refactoring, but API refactoring has its additional benefits:

Obsolete Programs Reuse Obsolete programs typically use platforms or libraries that are no longer maintained, and sometimes they tend to be unusable in the current hardware or software environment. Using these dated source codes is typically impossible unless an obsolete environment is constructed, and constructing such out-dated environment tends to be cumbersome and sometimes even impossible. In this case, API refactoring can enable programmers to reuse the obsolete source codes by refactoring the obsolete source code into a program that uses recently developed and accessible platforms or libraries.

Better Performance and Maintainability API refactoring can improve program execution performance if the underlying target library or platform has better performance. This is another important reason to perform API refactoring, especially when a better library or platform is available. In addition, performing API refactoring can improve code maintainability. This is especially true for the target libraries or platforms that provide some degree of domain specific language (DSL). The libraries that enhance graphical user interface (GUI) programming, for example, tend to include a DSL which makes GUI programming much easier.

There are two main difficulties of API refactoring: discovering semantically equivalent API calls between two libraries, and how to instantiate the new API calls using the environment's variables without changing the functionality. One of the earliest works [13] that aims to help API refactoring by using type constraints requires human interventions. It only focuses on refactoring *individual* API calls instead of refactoring sequences. Subsequent works have been focused on API translation or API mapping. Natural language features such as function names and comment descriptions can be

used to find similar API calls [14, 15, 16]. Direct API translations can also be learned using sequence alignment backed by statistical models, if a large number of API call sequence pairs that are semantically equivalent is available to serve as API translation examples [19, 17]. Researchers also applied API translations and migrations in a cross-language scenario [63, 17, 18].

In this chapter, we present a fully automated algorithm for API refactoring by using a combination of natural language techniques and program synthesis techniques. We do not consider using direct API translation examples to train a statistical model, because it requires a large amount of existing API translation examples which is difficult to harvest. Gokhale et al. [17] required users to exercise similar functionality in application pairs and collected semantically equivalent sequence pairs by analyzing execution traces. This data collection process becomes daunting when the goal is to develop a general-purpose algorithm for API refactoring; it requires people to repetitively exercise functionalities for potentially hundreds or even thousands of application pairs.

To illustrate how a programmer uses our algorithm to refactor an API call sequence, suppose s/he provides an API call sequence and the name of a target library. Our algorithm first uses a pre-trained language model to translate the given source API calls into a set of candidate API calls for the target library. This language model is trained using the comment text collected from all relevant APIs and also from the Java 8 API. For each set of translated API calls, the algorithm feeds the translated API calls into an API call sequence synthesizer called `Bayou` [1] which has learned all the API usage idioms from the source code of all relevant APIs. `Bayou` will synthesize a complete API call sequence that uses the target APIs.

We evaluated our algorithm on a diverse set of 37 benchmark problems which

contains API call sequences designed for everyday programming in the areas of I/O operations, GUI programming, networking, and document parsing and processing. The results from our evaluations show that our algorithm can refactor API call sequences with high accuracy where accuracy is defined as the largest length of the corrected generated API call sequence in the top-10 candidate solutions. In addition, our evaluations also demonstrate a high precision of our API translation method and also high robustness of the API sequence synthesizer we choose to use.

Here, our main contribution to the problem of API refactoring is as follows:

1. We propose the first fully automated algorithm that refactors API call sequences by combining natural language features and program synthesis techniques.

2. We present an implementation of our API refactoring algorithm and show that our algorithm is able to refactor API calls accurately in a variety of benchmark problems.

The rest of the chapter is organized as follows. Section 2 gives an overview of the problem and our method; Section 3 defines our problem formally; Section 4 describes our API refactoring algorithm; Section 5 presents our evaluations of our API refactoring algorithm and its individual components; Section 6 describes the related work. We conclude in Section 7.

3.2 Motivating Examples

In this section, we describe our API refactoring algorithm using a few examples.

A programmer has written a program that uses standard Java API to create an HTTP server showed in figure 3.1a. However, the user needs to update the program to use a library called `Grizzly` [64] which provides better performance and abstraction.

```
1   void create_server(int port,
2                      String url,
3                      com.sun.net.httpserver.
    HttpHandler h1,
4                      org.glassfish.grizzly.
    http.server.HttpHandler h2)
5       throws IOException {
6
7     //refactor:org.glassfish.grizzly
8     {
9       InetSocketAddress address = new
    InetSocketAddress(port);
10      HttpServer server = HttpServer.create(
    address, 0);
11      server.createContext(url, h1);
12      server.start();
13    }
14  }
15
```

(a) Java API

```
1   void create_server(int port,
2                      String url,
3                      com.sun.net.httpserver.
    HttpHandler h1,
4                      org.glassfish.grizzly.
    http.server.HttpHandler h2)
5       throws IOException {
6
7     HttpServer server = HttpServer.
    createSimpleServer();
8     ServerConfiguration config = server.
    getServerConfiguration();
9     config.addHttpHandler(h2, url);
10    server.start();
11  }
12
```

(b) Grizzly

Figure 3.1 : HTTP server refactoring example

Without using our algorithm, the user needs to search for relevant documentations and understand how to create an HTTP server using `Grizzly` framework. Then the user needs to copy and paste the example code snippets showed in the documentation and plug it into the current codebase by instantiating the API calls and using the predefined variables from the original environment as the arguments for the instantiated API calls. Refactoring this code snippet under a small project might be easy, but it will become much difficult when refactoring is done in a large software project.

However, our API refactoring algorithm provides a much easier experience of API refactoring. The user encloses the code snippet into a block and writes down the target library name as a comment, `org.glassfish.grizzly`, showed in figure 3.1a. Then the user provides the method that contains the API sequence block with the target library name and necessary variables which will be used in the target library such as `h2` to our algorithm as input. Our refactoring will pick up the method names such as `create`, `createContext` and `start` and their associated types such as `HttpServer` and understand that it needs to generate a program that uses `Grizzly` framework to create an HTTP server. In the end, it will return another method that contains another API call sequence showed in figure 3.1b which uses `Grizzly` and accomplishes the same task.

The output method is guaranteed to be compilable. Not only it automatically inserts necessary import declarations which are not shown here for brevity, but it also works with the existing context naturally; it produces the correct API calls in the correct order. In addition, the resulting API sequence uses the existing variables from the original environment as input arguments correctly, using types and additional hints from the API documentations as guidance. It also creates new objects such as `server` and `config` that the original method misses and adds necessary exception

```
1   void list_files(String username,
2                   String password,
3                   String host,
4                   String path)
5     throws IOException {
6     ...
7
8     //refactor:net.schmizz
9     {
10      FTPClient f = new FTPClient();
11      f.connect(host);
12      f.login(username, password);
13      FTPFile[] files = f.listFiles(path);
14      f.disconnect();
15    }
16
17    ...
18  }
19
```

(a) Apache Commons Net

```
1   void list_files(String username,
2                   String password,
3                   String host,
4                   String path)
5     throws IOException {
6     ...
7
8     SFTPClient sftpc1;
9     List<RemoteResourceInfo> l1;
10    SSHClient sshc1;
11    try {
12      sshc1 = new SSHClient();
13      sftpc1 = sshc1.newSFTPClient();
14      sshc1.authPassword(username, password)
    ;
15      sshc1.connect(host);
16      l1 = sftpc1.ls(path);
17      sftpc1.close();
18    } catch (TransportException _e) {
19    } catch (UserAuthException _e) {
20    }
21
22    ...
23  }
24
```

(b) SSHJ

Figure 3.2 : List files in an FTP server

handlings if necessary, which will be shown in the next motivating example. We emphasize that the user does not need to know how to create an HTTP server using `Grizzly` exactly, although s/he is required to provide necessary variables such as an HTTP handler named `h2` for the target library before refactoring.

Our algorithm is also able to refactor other API call sequences. Figure 3.2a shows an API call sequence that uses `Apache Commons Net` [65] to list files in an FTP server. Our algorithm can take this API calls sequence and the target library name suggested

```
1   void read(String filename)
2     throws IOException {
3     String text;
4
5     //refactor:org.apache.pdfbox
6     {
7       PdfReader reader = new PdfReader(
    filename);
8       text = PdfTextExtractor.
    getTextFromPage(reader, 1);
9       reader.close();
10    }
11
12    return text;
13  }
14
```

(a) ITextPDF

```
1   void read(String filename)
2     throws IOException {
3     String text;
4
5     PDFTextStripper pdfts1;
6     PDDocument pdd1;
7     File arg01;
8     try {
9       pdd1 = PDDocument.load((arg01 = new
    File(filename)));
10      pdfts1 = new PDFTextStripper();
11      text = pdfts1.getText(pdd1);
12    } catch (InvalidPasswordException _e) {
13    }
14
15    return text;
16  }
17
```

(b) PDFBox

Figure 3.3 : Read from a PDF document

and generate another sequence that uses `SSHJ` [66] and does the same task. Notice that our algorithm creates a new `SSHClient` object and a new `SFTPClient` object. It also adds checks to handle `TransportException` and `UserAuthException`, since they were not handled in the original input method.

Figure 3.3 shows another example where a programmer needs to refactor an API call sequence that uses `ITextPDF` [67] to read from a PDF document into another API call sequence that uses `PDFBox` [68]. Similar to the previous example, our algorithm creates additional necessary objects and exception handling checks. This is an example which demonstrates that our algorithm can refactor API call sequences for different application domains.

3.3 Problem Definition

Our system works with a subset of the Java programming language. We define T as a set of types and M as a set of methods. Each type has its constructors and methods and each method is defined as

$$t_o.m(t_1, t_2, \ldots, t_n) : r$$

where t_o is the type that owns the method, m is the method name, $t_1, t_2, \ldots, t_n$ are the types of the formal parameters and r is its return type. An API call c is defined as

$$v = o.m(a_1, a_2, \ldots, a_n)$$

where $o : t_o$ is an object and $a_1 : t_1, a_2 : t_2, \ldots, a_n : t_n$ are the input arguments of the API call. $v : r$ is an optional object that receives the return value of the call if r is not `void`. Then an API call sequence s is a sequence of API calls:

$$s : r_n = c_1, c_2, \ldots, c_n$$

We denote the type of a sequence s to be the return type of the last API call. We also define a function $ut(s)$ to be a finite set of types used in all API calls contained in s which includes the types that own the methods, all the argument types and all the return types.

The semantics of an API call sequence can be defined in the standard way with the exceptions that we do not allow generic types. However, we permit static calls, constructors and inheritance in our setting. If a call is a static call, an object o is not required. Constructors can be derived from changing the method name to `new` and the return type can be the owning type. Inheritance gives us a notion of subtyping relations between objects.

Let us define T_s as a finite set of source types, T_t as a finite set of target types, s_i as an input API call sequence under that $ut(s_i) \subseteq T_s$ and $t_r \in T_s \bigcap T_t$ as a requirement type. Our objective is to find another API call sequence $s_o : t_r$ under the condition that $ut(s_o) \subseteq T_t$ and that s_i is semantically equivalent to s_o.

3.4 Method

In this section, we present our API refactoring algorithm in detail. Our algorithm has two main subproblems: *API translation* and *API call sequence synthesis*. As we discussed in the introduction, we do not consider training a statistical model to learn direct API refactoring, because of the scarcity of training data. Generating a large number of pairs of similar API call sequences require humans to manually exercise similar functionalities and collect execution traces in hundreds of applications, which is a daunting task. We avoid such problem by breaking the API refactoring problem the two subproblems we mentioned above. In this way, we can make use of accessible resources such as Javadoc comments and API usages available in open source projects to still accomplish the task of API refactoring.

Algorithm 3 shows the overall algorithm. At line 1, we first translate the input API call sequence s into a set of candidate API calls S' using the API calls defined under the library k. Then at line 3, we use an API call sequence synthesizer to synthesize a candidate API call sequence solution s^* for each sequence $s' \in S'$. Finally, we return the sequence in which the return type is a subtype of the given requirement return type t. Next, we start describing our API translation method followed by the API call sequence synthesizer we choose.

Procedure 3 refactor

Require: An API call sequence, s, a target library k and a required return type t

Ensure: An API call sequence s^* containing only API calls defined in k

1: $S' \leftarrow \text{translate}(s, k)$
2: **for** $s' \leftarrow S'$ **do**
3: $\quad s^* \leftarrow \text{synthesize}(s')$
4: $\quad$ **if** match_type(s^*, t) **then return** s^* **end if**
5: **end for**
6: **return** *null*

3.4.1 API Translation

The first step is to translate a set of *individual* API calls from a library into another set of API calls defined in another library. Here, we borrow the idea from [14] and use natural language features as a glue to translate API calls. Given two sets of API calls across two different libraries, we translate the calls by calculating the pair-wise similarities. Specifically, when we try to translate a method m_1 from a library A, we calculate the *method similarity score* between m_1 and all the methods in library B. We choose m_2 from library B as the translation target if m_1 is the most similar to m_2 numerically. The similarity score between the two methods is defined as the weighted sum of the following three components. Figure 3.4 shows an example API translation:

Name Similarity The name of a method typically carries useful semantic information about the method. The name similarity is defined as the standard

cosine similarity between two word embeddings. To obtain word embeddings, we use `FastText` [69], because `FastText` allows us to generate meaningful and accurate embeddings even for CamelCase words such as `BinarySearch` and `GetInputStream`.

Type Similarity We treat types as natural language words in this case. Similar to name similarity, the type similarity is defined as the cosine similarity between two word embeddings using `FastText`.

Description Similarity The description of a method is a list of sentences extracted from the Javadoc comment. We use `BERT` [70] because it can capture contextual information about words, in addition to semantic meaning. The similarity between two method description can be derived by calculating the sentence similarity.

We give more weights to description similarity and less weights to the rest because in most cases descriptions carry the most important information. We only use the first two sentences of the description if there are more than two sentences, because the first two sentences are the most important and typically the rest of the text is misleading. We also set a similarity threshold in order to prevent meaningless translations. If two words are not similar enough, we will discard the translation. This is necessary to prevent unreasonable translations especially when two libraries have different terminologies and coding practices. We will discuss this issue later in the evaluation section. Finally, we extract the text from the Javadoc of all the relevant methods along with the text from Java 8 standard library Javadoc to train `FastText` and `BERT`.

Name: clean
Type : TagNode
Desc : Parse an html document string

(a) HtmlCleaner's parse function

Name: parse
Type : Document
Desc : Parse HTML into a Document.

(b) Jsoup's parse function

Figure 3.4 : API translation example

3.4.2 API Call Sequence Synthesis

The next step is to synthesize complete API call sequences using a set of translated API calls. Here we use *Bayou* [1] to synthesize API call sequences. *Bayou* is a conditional program generation system. Given an *evidence* containing names of API methods and relevant types, it is able to synthesize a set of complete API call sequences that use the API calls or relevant types suggested in the provided evidence.

Figure 3.5a shows an example program where the goal is to log into an FTP server and list the files in a given path. The evidence here contains call names such as `connect` and `ls`, indicating that the user needs to extract the file names in the given path. *Bayou* takes this evidence along with the variables in the environment and generates some candidate API call sequences. One example candidate sequence is showed in figure 3.5b.

Bayou works by learning over program sketches which abstracts out noises such as concrete names and operations. It uses a probabilistic encoder-decoder called Gaussian Encoder-Decoder to learn a distribution over program sketches conditioned on evidences. To synthesize a program given an evidence, it samples sketches from the distribution using the provided evidence. Then it uses combinatorial search to

```
1   void list_files(String username,
2                   String password,
3                   String host,
4                   String path)
5     throws IOException {
6     {
7       ///call:close call:ls
8       ///call:connect call:authPassword
9     }
10  }
11
```

(a) Evidence for FTP file listing

```
1   void list_files(String username,
2                   String password,
3                   String host,
4                   String path)
5     throws IOException {
6
7     SFTPClient sftpc1;
8     List<RemoteResourceInfo> l1;
9     SSHClient sshc1;
10    try {
11      sshc1 = new SSHClient();
12      sftpc1 = sshc1.newSFTPClient();
13      sshc1.authPassword(password, password)
    ;
14      sshc1.connect(host);
15      l1 = sftpc1.ls(host);
16      sftpc1.close();
17    } catch (TransportException _e) {
18    } catch (UserAuthException _e) {
19    }
20  }
21
```

(b) Sketch example generated by Bayou

Figure 3.5 : API translation example

concretize the sampled sketches into type-safe programs.

To use *Bayou* in our scenario, we first train *Bayou* using the training data gathered from the source code of all relevant libraries and methods. The training data contains a large amount of program-evidence pairs where the evidence consists of the names of the method calls in the corresponding program. This training data associate the evidences with the API call sequences used in the corresponding program. To synthesize a candidate API call sequence, we feed a set of translated API calls s' to *Bayou*. *Bayou* will sample a set of sketches from the learned distribution, concretize the sketches and returns the synthesized programs consisting of the target API call sequences. The input program in figure 3.5a shows an example translated API calls such as `connect` and `ls` and figure 3.5b shows the concretized program generated by *Bayou*. *Bayou* typically synthesizes multiple candidate programs. To generate a correct solution, we enumerate all the candidate sequences and return the first top-k programs where the return type is a subtype of the requirement type.

One feature of *Bayou* is that it guarantees to generate type-safe programs, as their paper suggests [1]. However, while using `Bayou` to synthesis API call sequences, we notice that `Bayou` sometimes fails to generate arguments, especially for the arguments that have the same type. Figure 3.5b shows such an example, where it fails to generate correct arguments for method calls such as `authPassword`, and `ls`. Although `Bayou` is able to produce *type-safe* API call sequences, it tends to fail when types cannot help.

To solve this problem, we create *semantic models* for variables by using the libraries' source code. These semantic models can guide us to generate correct arguments. Figure 3.6 illustrates the argument matching method using an example which tries to refactor an API call for logging into an FTP server. In the first step, we

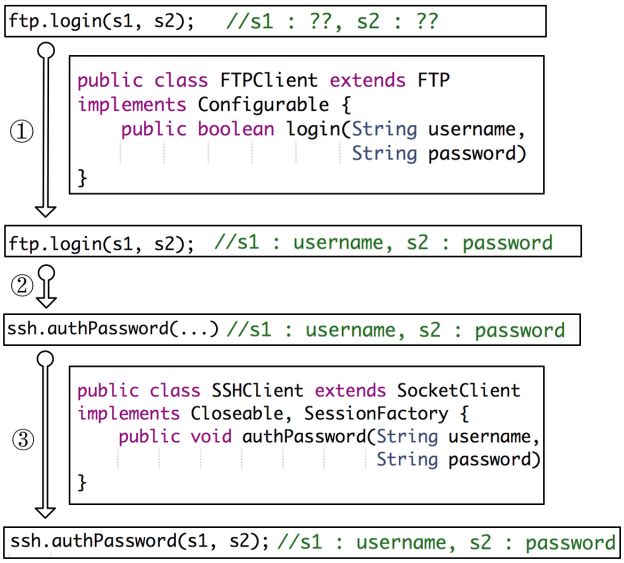


Figure 3.6 : Argument matching

do not know anything about the variables `s1` and `s2`, but we can inspect the formal parameters of the method `login` through its method call. From the source code, we can construct semantic models for `s1` and `s2` using relevant natural language features such as their formal parameter names, `password`, and `username`. The second step involves normal API translation step which translates `login` into `authPassword`. In the third step, we can inspect the source code of `authPassword` and match the natural language features of the formal parameters to the semantic models of `s1` and `s2`. Finally, we can use `s1` and `s2` as the first and the second arguments respectively according to the matching result. Notice that a semantic model does not have to use formal parameter names or even natural language features as the primary driver. Anything that has semantic information would suffice.

```
//refactor:net.schmizz
{
  FTPClient f = new FTPClient();
  f.connect(host);
  f.login(username, password);
  f.retrieveFile(path, out);
  f.disconnect();
}
```

(a) Input for FTP upload

```
SSHClient ssh = new SSHClient();
SFTPClient ftp = ssh.newSFTPClient();
ssh.authPassword(username, password);
ssh.connect(host);
ftp.get(path, filename);
ssh.disconnect();
```

(b) Output for FTP upload

Figure 3.7 : FTP upload benchmark problem

3.5 Evaluation

In this section, we evaluate the performance of our refactoring algorithm by conducting a series of experiments. These experiments consist of running our refactoring algorithm on various inputs and test how well the algorithm would produce the expected API sequence. In addition, we evaluate the performance of our API translation component to test if it can translate API calls accurately enough for the API call sequence synthesizer to produce correct sequences. Then we see if *Bayou*, the API call sequence synthesizer we choose, is a good choice for API refactoring, by testing its robustness given translated API calls in various degree of accuracies.

3.5.1 Benchmarks

Our benchmark problems consist of refactoring API call sequences from various domains including CSV handling, networking, document processing, and machine learning. For each benchmark problem, our algorithm needs to refactor an input API call sequence into another sequence using the API calls defined in another library. Figure 3.7 shows an example benchmark problem where the input program uploads a file

```
1   //refactor:org.apache.pdfbox
2   {
3     PdfReader reader = new PdfReader(
    filename);
4     String text = PdfTextExtractor.
    getTextFromPage(reader, 1);
5     reader.close();
6   }
7
```

(a) Input for PDF read

```
1   File infile = new File(filename);
2   PDDocument document = PDDocument.load(
    infile);
3   PDFTextStripper stripper = new
    PDFTextStripper();
4   String text = stripper.getText(document);
5   document.close();
6
```

(b) Output for PDF read

Figure 3.8 : PDF read benchmark problem

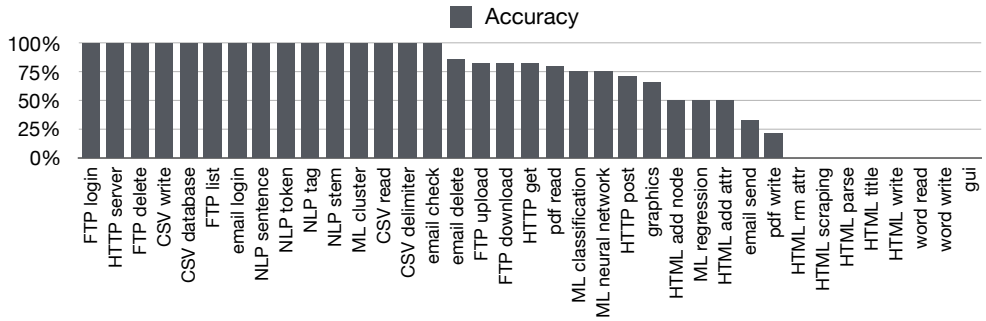
to a remote FTP server. The enclosing context which includes variable declarations and parameters are omitted for brevity here. Figure 3.8 shows another example where the goal is to get the text as a string from a given PDF document.

To test the performance of our refactoring algorithm quantitatively, we define *refactoring accuracy* as the highest percentage of the correct API calls generated in the top-10 programs. A refactoring algorithm that achieves high accuracy in our benchmark problems is considered effective.

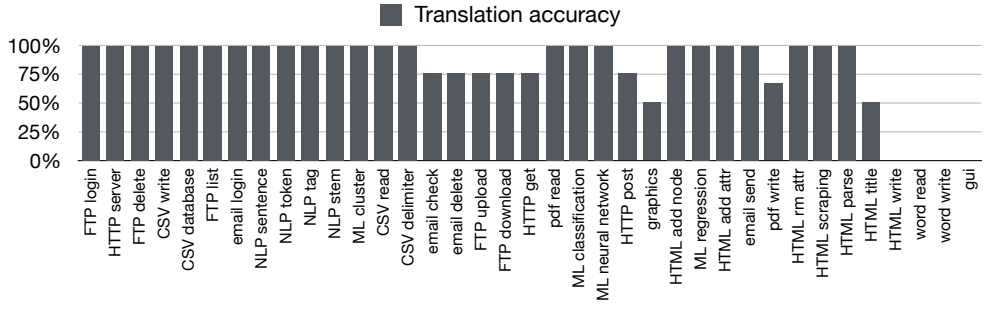
3.5.2 Experiments

We implemented our algorithm in Scala 2.12.3 and 64-bit OpenJDK 8 and the algorithm runs on a 2.2GHz Intel Xeon CPU with 40 cores with 500GB RAM. For each benchmark problem, we record the highest refactoring accuracy among the top-10 programs. We also set the time limit to be 10 minutes and we abort any run that exceeds the time limit. Figure 3.9a shows the accuracies of the runs on all the benchmark problems with and without considering argument correctness.

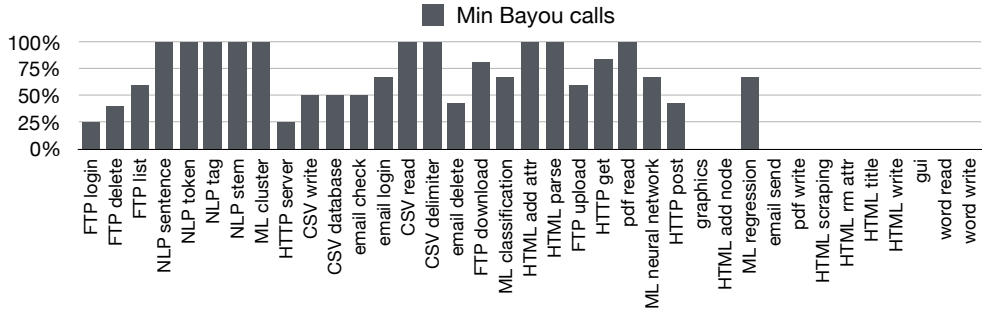
According to the results, we can see that our refactoring algorithm is quite effective



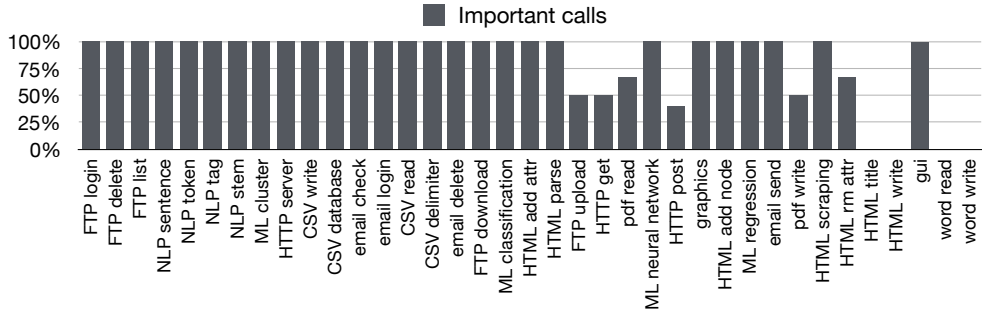
(a) Overall refactoring accuracy



(b) Translation accuracy considering top-5 results



(c) Minimum calls required for Bayou to synthesize correct sequence



(d) Number of important calls fed to Bayou during overall refactoring accuracy evaluation

Figure 3.9 : Evaluation results of our overall algorithm and each individual component

```
1       BodyContentHandler handler = new
        BodyContentHandler();
2       TikaConfig tika = TikaConfig.
        getDefaultConfig();
3       Metadata metadata = new Metadata();
4       Parser parser = tika.getParser();
5       ParseContext context = new ParseContext();
6       parser.parse(stream, handler, metadata,
        context);
7       String text = handler.toString();
8
```

```
1       XWPFDocument docx = new XWPFDocument(
        stream);
2       XWPFWordExtractor we = new
        XWPFWordExtractor(docx);
3       String text = we.getText();
4
```

Figure 3.10 : Reading from a Word document

and accurate. It can successfully refactor API sequence as long as the API translation achieve high accuracy. In some cases such as `Checking Email` and `FTP download`, even though our API translation result is not perfect and cannot translate all API calls, the API sequence synthesizer can still infer correct API call sequence and produce the right output.

However, our refactoring algorithm can fail in the following cases:

Different terminologies Sometimes different libraries might have different coding practices and methodologies. Figure 3.10 shows such an example. In `Word Reading` problem, two libraries, `Apache Tika` and `Apache POI`, are used for extract text from a Microsoft Word document. `Apache Tika` requires objects such as a `BodyContentHandler`, a `TikaConfig`, a `Metadata` and etc., whereas `Apache POI` only requires `XWPFWordExtractor`. This leads to a significant difference in their Javadoc comments in terms of terminologies. As a result, our API translation algorithm fails to find semantic similar words, which leads to translation failure. In short, our API translation cannot work with libraries

that have different vocabularies.

Rare sequences Some API sequences such as `open-read-close` are quite common in the data used to train the API sequence synthesizer. Typically the synthesizer can easily generate these sequences. However, the synthesizer tends to fail to synthesize rare sequences in some benchmark problems such as `Sending Emails` and `PDF Writing`, because *Bayou* fails to learn those sequences. This is one of the common problems with corpus-driven program synthesis [71].

API translation evaluation

It is important to study the overall accuracy of our API translation method, because the synthesis step depends on the quality of API translation heavily. Here the accuracy is defined as the percentage of corrected translated API calls in top-k translations. Since all translations are based on calculating similarities between words, we collected the text from all the relevant APIs' Javadoc comments and also from the Java 8 standard API. We then used all the text and trained `FastText` and `BERT`. The goal is to have a statistical model that allows us to calculate similarities between terms specific to Java programming. Table 3.1 shows ten similar words associated with some selected query words. We can see from the table that our `FastText` model seems effective in grouping the words that have similar semantic meaning.

Figure 3.9b shows the accuracies of translating API calls across two libraries. Given an API call, our algorithm outputs multiple candidate translations ordered by similarity descendingly, and we define accuracy as the percentage of correctly translated API calls appearing in the top-k results. We plot the accuracies when we consider top-5 API call translation results. We can see that for most benchmark problems, our method is quite accurate; in most cases, correct translations appear in

Query word	Similar words
`int`	`integer`, `float`, `long`, `double`, `short`
`ftp`	`nntp`, `smtp`, `secret`, `pixmap`, `out-of-synch`
`button`	`rollover`, `radio`, `tooltip`, `checkbox`, `click`
`index`	`IndexFrom`, `MenuIndex`, `ListIndex`, `occurrence`, `nth`
`stream`	`InputStream`, `StreamB`, `BufferTest`, `console`, `AccessFile`
`image`	`gif`, `animation`, `texture`, `BufferedImage`, `RenderedImage`
`email`	`bcc`, `recipient`, `sender`, `addresse`, `mail`
`vector`	`scalar`, `dense`, `product`, `kernel`, `matrix`

Table 3.1 : Similar words generated by FastText

the first three results. However, our translation fails for some benchmark problems such as `Word read` and `Word write`. The main reason is that the terminologies and coding practices across these libraries are different; it is hard to establish API call mappings between them. Figure 3.10 shows two programs that read from a Word document. One program requires objects such as `Metadata`, `Parser`, `ParseContext`, while the other requires `XWPFDocument` and `XWPFWordExtractor`. It is impossible to translate concepts like `BodyContentHandler` and `Metadata` across two programs.

Synthesizer evaluation

Having evaluated our translation method, let us switch to the API synthesizer we use, *Bayou*. The main question we need to answer here is: is *Bayou* a good choice for synthesizing API call sequences in our setting, given various translation inputs? It is important to understand how robust *Bayou* is and when would *Bayou* fail. If we

provide fewer API calls to *Bayou*, will it still be able to synthesize the correct API call sequence?

Figure 3.9c shows the proportions of translated API calls we need to feed into *Bayou* in order to have the correct API call sequence. 0% means *Bayou* cannot synthesize the correct sequence given all necessary API calls. From the figure we can see that even if we omit a small portion of calls, *Bayou* is still able to synthesize correct sequences, indicating that *Bayou* is seemingly robust. However, this begs another question: which API calls can we omit? As we tried different combinations of API calls to feed *Bayou*, we discovered that *we can omit API calls that serve as bases for other API calls*. For example, figure 3.7a shows a program that downloads a file from an FTP server. The gist of the calls in this program is `retrieveFile` and `disconnect`, as `retrieveFile` depends on `connect` and `login`; *Bayou* can easily infer the basis API calls from `retrieveFile`.

This result indirectly leads to another question: is *Bayou* able to synthesize correct API call sequence when we give only important calls and ignore the bases? We recorded the numbers of important calls fed into *Bayou* during the runs for the benchmark problems and figure 3.9d shows the result. As we can see that for the cases where all important calls are provided it is very likely that the final synthesis accuracy will achieve at least 75%. If at least one important call is missing, the accuracy tends to drop significantly. One exception is the `GUI` problem where redundant and noisy calls are fed into *Bayou* which also causes the failure.

Bayou also fails to synthesize correct sequences when the input API calls cannot help identify the correct sequence. In `HTML title` problem where the goal is to parse an HTML string and call `title` to get the title of the HTML document, *Bayou* cannot use the calls such as `title` to synthesize correct sequences, since `title` is widely used

in many other tasks which provides little information. Another reason why *Bayou* might fail is that the particular API sequence we need to synthesize is quite rare in the training data as we discussed above. In the `graphics` benchmark problem, we discovered that sequences which contains a `drawRect` call that followed by another `drawImage` call are quite rare in our training data.

3.5.3 Limitations

Finally, we summarize the limitations of our algorithm:

1. Our algorithm relies on API translations as the first step. However, the API translation method will fail if the terminologies vary dramatically across two libraries. These libraries tend to use words that have different semantic meanings to describe the same functionality, which prevents accurate API translations.

2. We use *Bayou* to synthesize complete API call sequences by taking individual translated API calls as input. *Bayou* learns API usage idioms from relevant source code as the training data, but sometimes it fails to generate the target sequence that does not appear in the training data.

3.6 Summary

Software refactoring has been a cumbersome task and also is as difficult as software development. Meanwhile, code refactoring plays an important role in maintaining dated programs and also in improving existing codebase in terms of functionality and performance. In this paper, we propose an algorithm for API refactoring, which refactors an API call sequence into another without modifying the functionality. We first translate the input API call sequence into a set of individual API calls defined

in the target library. Then we use an API sequence synthesizer to generate a complete API call sequence. We evaluated our algorithm on a diverse set of benchmark problems. The results showed that our algorithm can refactor API sequences with high accuracy.

Chapter 4

Related Work

4.1 Program Synthesis and Reuse

The problem of program synthesis has been considered for a long time. Pnueli and Rosner studied the problem of synthesizing [28] for reactive systems where they developed an algorithm for synthesizing a finite-state reactive program satisfying a given specification and a better algorithm for checking the emptiness of automata on infinite trees. Jobstmann et al. [72] considered consider the problem of program repairs as a game by turning an LTL game into a program game by freeing the values of expressions. Finally, they showed a polynomial-time heuristic for generating memoryless strategies. People used deductive and SMT Solvers for program synthesis. Manna and Waldinger [24] introduces a deductive based theorem-proving technique for program synthesis where they presents a series of deduction rules. Torlak and Bodik[73] presents a symbolic virtual machine (SVM) which compiles to constraints using only a small subset of a DSL designed by the user. They use type-driven state merging to solve the problem of exponential state blowup, and it provides an interface for DSL designers to extend symbolic evaluation to unlifted operations.

Providing additional constraints has been proved to be effective in reducing the search space. SKETCH [7] uses SAT solves and counterexample-driven method to synthesize constant values in a finite and partial programs. SKETCH also uses a subset of C grammar, but our method could synthesize blocks of code instead of

constant values. Syntax-guided synthesis [29] uses CEGIS and syntax to guide the search and they have showed some successes in synthesizing reasonable-sized and correct programs. Kuncak et al. [74] turns a decision procedure into a synthesis procedure for quantifier-free formulas for various theories. Similar to our method, template-based method [8] uses programmer-defined template to guide the search for verification and synthesis. However, they convert the template into a set of constraints to do synthesis.

Synthesis using inductions and examples has been studied for a long time. THESYS [75] is an automatic program system for synthesizing recursive LISP programs from examples. Kitzelmann and Schmid [76] extends the previous work by developing a method which can induce multiple recursive equations in one synthesizing step. Using type to guide the search space has been showed to be quite effective [30, 77]. Induction-based synthesis has also been applied to many kinds of application domains including geometry construction problems [10] and spreadsheet data manipulation [9].

Code Conjure [54] was one of the earliest works that considered the problem of code reuse. It works by first search for relevant software components from a search engine which supports a special query language designed specifically for code search. Then it uses the user-provided tests to select the best software components into programmers' working environment. This idea is very similar to our program splicing method, but Code Conjure only works at the granularity of functions. CodeGenie [55] proposed a similar idea using a better code search engine called Sourcerer [45]. They presented an implementation as an Eclipse plugin. Reiss [53] proposed a similar method for code search as well, but with more emphasis on code search. We will discuss their methods in detail in section. In summary, the methods mentioned here consider reuse programs at the granularity of functions. On the other hand, our

method provides a Sketch-like interface where programmers can leave holes. Our method allows reuse at the level of statements and expressions, making it more likely that reuse is applicable.

Gilligan [78] proposed a more advanced method for code reuse. It captures the developer's intent by using a programmatic-reuse plan. This programmatic-reuse plan helps Gilligan migrating the external source code into the developer's working environment by determining the source code component for reuse and adding and deleting source code necessarily. They evaluated their method by conducting a user study which demonstrated that their tool could save more time. Our idea of program splicing is very similar to Gilligan, except that we use combinatorial search combined with tests to integrate external code snippets.

Narasimhan and Reichenbach also considered the problem of copy-and-paste [79] and they seeks to redeem copy-and-paste through a method that finds clones of a fully written program and automatically merges these cloned code with the new code. However, their work only considers extremely similar programs whose parse trees are a few edits away and does not consider draft programs. In contrast, our approach can bring arbitrary programs from the internet into a programming context, and uses a combinatorial synthesizer to splice this code into the context.

Genetic programming has also been used for code transplantation. μ-Scalpel [32] is a well known tool that uses genetic search to transplant a software component into software project. μ-Scalpel uses genetic programming to search for relevant code fragments and filters out the undesired ones by using user-provided tests. They have showed that they are able to transplant major software components into one another. Subsequent works focused on applying the idea to real-world projects such as Pidgin [80], Kate [81] and helping citation services [82]. Genetic programming

has also been applied to improve the performance of C++ programs with the help of a Boolean satisfiability (SAT) solver [83]. Although genetic programming has been showed to be effective in code transplantation, it suffers from efficiency problem, as we discussed in section 2.5 where it fails most of our benchmark problems.

CodePhage [33] transplants arbitrary code snippets across different applications, but it focuses on applying patches to existing buggy applications where the patches are generated from external source code. It first identifies the branches in the applications that triggers errors. Then it search for candidate donors and insertion points that potentially eliminate the errors. Finally it validates the patches by running regression tests. The idea of CodePhage is very similar to our splicing method, but CodePhage exclusively targets to applying patches for binary programs. It is unknown if its application could be broader and if it can be used in the source code level.

4.2 Data-driven Program Synthesis

With the booming of online source code repositories and the popularity of Massive Open Online Courses (MOOCs), some interesting work has been focused on the tasks involving dealing with a large amount of program source code. Hindle et al. [39] studied a large amount of source codes available online and discovered that source codes written by real programmers are highly repetitive and predictable. Therefore, they found out that statistical language models can capture local regularities in real-world softwares. They indicated that those learned models can be used for code completion and prediction.

Mishne et al. [20] focuses on mining the specification for API calls from a large amount of code snippets. It first collects the source code from popular source code repositories such as `GitHub` and `SourceForge`. Then a large number of partial speci-

fications are derived from small code snippets from those source codes. Finally, complete API specifications can be consolidated from those partial specifications. They also showed that those API specifications have been showed to be useful in several tasks, including API call predictions and static analysis.

Online source code can also be used in general static analysis. JSNice [2] is a popular tool that deobfuscates `JavaScript` programs by renaming variables and inferring variable types. It works by using conditional random fields (CRF) to learn program properties from a corpus of `JavaScript` source codes. These properties typically consist of variable types and names. After a CRF finishes learning those properties, it can be used to predict variable names and types. They also evaluated their model and showed the effectiveness of their method. In the context of MOOC, a large amount of source code has been showed to be effective in grading graphical programs with high-level specification [84]. The idea is to first learn hidden specifications from a large amount of syntactic and semantic features of highly scored programs, and then we can grade another unseen program by comparing the unseen features with the learned specifications.

Raychev et al. [38] particularly focuses on learning programs from noisy data. They propose a combination of a regularized program generator and a dataset sampler to create a feedback loop which can detect errors and avoid overfitting. They have showed that their method is very effective in synthesizing bitstream programs using incorrect program examples and also in dealing the growth of new examples. In addition, they also design a domain specific language (DSL) for a new code completion system which has been showed to be more flexible and generalizable compared to other existing code completion methods.

Using statistical methods to help API call prediction has been showed to be very

effective. Traditional language model such as N-Gram and recurrent neural networks (RNN) can be learned from a large amount of API call sequences [21]. N-Gram model can be more effective in predicting short sequences, whereas RNNs is better at predicting long sequences. Baysian model [22] can also be used to learn API specifications and detect API usage anomalies. In addition, Murali et al. [1] also have showed that deep learning can also be used to learn API specifications which in turn can predict API call sequences. The novelty in this work is that it learns abstract representations of API call sequences instead of concrete sequences. They have showed that these abstract representations can also be used for API call predictions. GraLan [85] is a graph-based statistical model for code suggestion. It represents code snippets by analyzing abstract syntax trees (AST) and constructing graphs which can capture control flows better than traditional language model such as N-Gram. They have showed that these graph-based models can be effective in predicting API calls and small program expressions.

Deep learning has gained much attention in recent years and DeepCoder [23] is one of the earliest work which shows deep learning can help program synthesis. They design a small DSL that can represent common data processing and manipulation tasks. After they have programs expressed in the DSL, they can train their RNN by generating a large number of input-output examples and feed those examples into the neural network. They have showed that their RNN can speed up the synthesis process by an order of magnitude. However, their method is limited to synthesizing small programs; they have not showed that their model can synthesize larger programs.

Online code corpus can also be used for program repair and patch generations. Prophet [86] focuses on program repair by using a set of program transformations learned from a large amount of existing patches gathered from online repositories.

These transformations include inserting/deleting statements and guards. Then it uses the derived transformations combined with Staged Program Repair [26] to generate and validate candidate patches. They have showed that their method can generate patches for real-world applications. CodePhage [27], as we discussed in section 4.1 transfers correct code from donor applications into a buggy application.

Similar to our splicing work, some researchers have also tried to use a combination of natural language and program snippets to predict code snippets and API calls. SWIM [34] first takes a user query which contains English words indicating the API of interest. Then it uses the clickthrough data gathering from well-known search engine such as `Bing` to look for relevant API code snippets and represent them using a special structure which captures simple control flows such as branches and loops. They have evaluated their method using 30 API related queries and their method can produce relevant solutions in top-10 results in all benchmark problems. `AnyCode` [57] has very similar idea, but it uses a probabilistic context free grammar to represent Java constructs and method calls and a customized natural language processing engine to extract informations from queries. They have showed that their method is flexible in handling queries and it is able to repair incorrect Java expressions. These two works are very similar to our program splicing, but SWIM and `AnyCode` only focuses on generating API code snippets whereas our splicing method is more general.

4.3 Code Search

The purpose of code search is to quickly search a large code corpus for the source code that is relevant to a task. It plays a significant role in data-driven program synthesis. Roy et al. [87] have given a comprehensive survey of various techniques for similar code search and code clone detection. They classified those techniques into

four categories: *textual*, *lexical*, *syntatic* and *semantic*. We will next introduce these techniques briefly.

Textual-based techniques treats program source code as strings. Comparisons between programs are done directly on those strings with little or even no preprocessings. Therefore, textual-based methods can extract "fingerprints" from program source codes and they are often used to detect exact matches. Lexical-based methods typically use parsers to extract tokens from programs. Then comparisons are done on the extracted token sequences. Programs with duplicate subsequences are considered as similar or duplicates. Compared to textual-based methods, lexical-based techniques can handle simple formatting and spacing issues during code comparisons. Syntatic-based methods typically require input source code to be parsed into abstract syntax trees (AST). Then tree matching techniques can be used to compare those ASTs. Sometimes structural metrics are defined to help the comparisons. Syntactic-based methods are typically more robust than lexical-based methods in dealing with noises such as positional changes. Finally, semantic-based methods use static analysis on input programs to extract semantic informations which will later be used for comparisons. Program dependency graphs and types are usually extracted as semantic informations.

`DECKARD` [88] is a scalable and efficient code clone detector that uses syntactic approach. It extracts syntax features from source code and represents those features using real vectors. The programs along with their feature vectors will be clustered together, and the programs in the same cluster are consider similar. They presented an implementation of their method called `DECKARD` and they have showed that their method is able to detect code clones in C and Java. Keivanloo et al. [89] supports free-form queries for similar code search. It works by first construct a database

of programs with their extracted keyword sequences. Code search is done by first accepting a free-from query from the user which will be translated into a set of query keywords. These query keywords will be used for code search in the end. They have showed that their method can be integrated into existing web-based code search engine and that it outperforms existing web-based code search engines. Our code search technique in the splicing work also supports free-form queries, but our search technique depends on comparing natural language features instead of syntactic and semantic features from source codes.

Reiss [53] proposed a code search engine that allows users to specify various semantic constraints. These semantic constraints include function signatures, security constraints and input-output constraints. Reiss presented an implementation of the code search and showed that using a combination of these constraints along with post-processing can be effective in locating program components that were not accessible using traditional code search engine.

One of the drawback of Reiss's method is that sometimes it might be difficult to specify some semantic constraints. Stolee and Elbaum [90] improved on semantic code search by using SMT solvers which accepts lightweight specifications. Compared to Reiss's method, those lightweight specification allows users to to specify semantic constraints more easily. Code search is performed by first transforming a large collection of programs into constraints offline. Users can then search for interesting programs by providing constraints. Finally, the search engine will use an SMT solver to locate programs that satisfy the input constraints. They have showed that using lightweight constraints and SMT solvers can be effective in search for simple programs. Ke et al. [91] applied similar code search method to program repair. To repair a buggy program, Ke et al. proposed to extract patches from correct and semantically similar

programs and use the extract patches to repair the buggy program. Their code search technique is very similar to Stolee and Elbaum's technique. They first construct a database of functions along with their input-output examples. Then similar functions can be located by comparing the input program's input-output examples with the ones in the database.

`TRACY` [92] is another similar code search method that uses *tracelets*. Tracelets are a collection of partial control-flow graph extracted from functions that begins and ends at control-flow instructions. After the function are decomposed into sets of tracelets, similarities are computed by counting the number of rewrite steps from one tracelet into another. `TRACY` has been applied to searching similar function in binary, and it has better precision and recall than existing techniques such as n-gram and graphlets. `TRACY` is different from our code search technique in a way that we only targets programs in the source level instead of binary level.

Kashyap et al. [40] proposed a hybrid method for code search. This method uses a combination of syntactic and semantic features including relevant data types, names, function signatures and abstract tree structures. It also supports searching using natural language features such as tf-idf. The search engine is flexible in a way that users can choose to use specific subsets of features and assign different weights as importance indicators to features. They have demonstrated improvements on precision and recall over existing code search techniques. In our program splicing work, we use this method as our main code search technique. However, we only use natural language features to search relevant programs instead of using syntactic or semantic features.

Grechanik et al. [93] also proposed *Exemplar* for code search using natural language. *Exemplar* works by first analyzing help documents, relating natural language

keywords to corresponding API calls and building connections between those API calls and the applications where the API calls are used. When a free-form query is provided by the user as input, *Exemplar* uses the keywords from the input query to look for relevant API calls followed by the related applications. They conducted a user study with 39 professional programmers and have found that *Exemplar* outperforms existing code search engine such as `SourceForge`. *Portfolio* [94] is another similar code search engine. It takes a free-form query from the user as input and search a database of analyzed functions with metadata for relevant functions. The difference is that *Portfolio* combines natural language processing (NLP) techniques, such as stemming and identifier splitting, and indexing techniques such as *PageRank* and spreading activation network (SAN) algorithms. *Portfolio* has also been showed to be more accurate in code search than existing engine such as `Google Code Search` and `Koders`. Similar to the code search technique we use in our program splicing work, these techniques all support free-form queries from users. The difference here is that our code search technique only relies on calculating the importance of keywords.

4.4 API Refactoring and Translation

API refactoring and migration has been studied for more than a decade. Balaban et al. [13] proposed an automatic method for migrating applications that uses obsolete libraries. Their method requires a *migration specification* which defines how to map obsolete API usages into their replacements and their method is able to use *type constraints* to determine where a refactoring can be applicable. Their evaluation shows their method is able to rewrite obsolete API usages in some real-world applications. However, their refactoring algorithm cannot handle API call sequences. Tayton et al. [95] mined well-known library repositories such as Maven Central, Google Code,

SourceForge and Github and produced library migration graphs to help library migrations. Their method is able to detect common library migrations, but unlike our API refactoring method they can only suggest new libraries instead of automate the process of actual migration.

API mapping has also been studied and one important application is to map API across different languages such as Java and C#. Natural language features such as names and descriptions can used as a glue to relate similar API calls. Zhong et al. [63] proposed a method to map APIs from Java to C#. They map the classes and methods in the client code that contains API usages from Java and C# by using class and method names along with *API transformation graphs* to map methods as well as their parameters and return values. In contrary, our API translation method mostly relies on comments from methods and an API sequence synthesizer. Finally, their experiment shows that their method can indeed help project migration from one language into another.

Pandita et al [14] uses natural language to mine API mappings across different platforms. They map APIs in different libraries using method descriptions, type names, method names and etc. and they use text embedding models to represent words and map APIs by calculating similarities between words. They showed that they are able to map more APIs in different platforms than Rosetta [17]. We borrowed the API mapping method from this paper for our API refactoring algorithm, but in contrast our goal in API translation is to map similar methods from two libraries in the same language. Text embedding model can also be useful in finding API usage relations [15, 96] and API mappings can then be inferred using two sets of similar API usage patterns. However, they did not show that using API usage relations could generalize to be useful in mapping the APIs that do not have obvious usage patterns.

Aura [16] solves one-to-many and many-to-one API mapping problems particularly using natural language similarity with additional call dependencies. Their method has better precision on several real-world applications and their multi-iteration algorithm allows them to map one-to-many and many-to-one change rules. However, our API translator does not try to solve one-to-many and many-to-one problem.

Sequence alignment is one promising research direction for API mapping or translation. Gokhale et al [17] infers API mappings by analyzing API call trace pairs generated by running similar pairs of programs. They use *Factor Graphs* to represent all possible API mappings which allows them to infer the most likely API mappings. Their method is able to map one API call into a sequence of API calls and has overall good accuracy. Our API translation method is different in the way that we mostly rely on natural language and we do not consider the one-to-many mapping problem. Program paths [18] can also be treated as *sentences* where individual API calls are treated as *words*. As a result, API mapping can then be considered as a natural language translation problem where the goal is to find the mappings between words in two different sentences generated from two different libraries. They gathered apps for Android and iOS platforms to evaluate the performance, but it is still unclear what the actual result is.

StaMiner [19] constructs *Groums* [97], which are graph representations for API usages, from existing source code and extracts API usage sequences called *sentences* from Groums. Then they use a Expectation-Maximization (EM) algorithm to align two API sequences and the alignment that has the highest score serves as the result of API mapping. Their method can achieve high accuracy and can also handle one-to-many and many-to-one problem naturally. Our API translation method is different in the way that we do not gather API call sequences for mapping APIs.

API usage patterns and specifications have also been showed to be useful to mine similar APIs. Nguyen et al. [98] considers the problem of software evolution adaptation. They mined adaptation patterns from existing softwares in two different versions and use graph-based object model to recommend actual adaptations. Their evaluation shows that their method is able to mine adaptations accurately, but it is unclear if users need to perform the actual software adaptations. Zhong et al. [41] tries to mine resource specification from API documentations. They infer resource specifications from method descriptions and use automata as representations which allow them to detect bugs where the specification is violated. They evaluated their method and showed that their method is able to infer specifications accurately. However, it is unknown to us that whether their method could generalize to more open source libraries.

Chapter 5

Conclusion and Future Work

Software development is difficult and time-consuming. Even though researchers have been developed programming tools for more than a decade, the performance of traditional programming tools tends to suffer due to their scaling problem. With the advent of "big code", researchers have been able to mitigate the scaling problem significantly and develop advanced and novel data-driven algorithms for programming tools. The idea of "big code" has been driving the technological frontier for programming tools forward, especially in the areas of program analysis and program synthesis.

By using "big code", we have proposed two programming systems to facilitate two common software workflows: software reuse and software refactoring. We first proposed *Program Splicing* which automates the process of copying and pasting using a code corpus. Our evaluation demonstrated that Splicer can outperform a state-of-the-art programming system. Our user study also showed that Splicer can boost software reuse productivity significantly. Second, we proposed a fully automated algorithm for *API refactoring* which relies on API translation and API sequence synthesis. We showed that our API refactoring algorithm is quite accurate in refactoring API sequences given that the coding styles and the terminologies are similar across two libraries.

Even though the evaluations of our techniques introduced in this thesis are quite optimistic, we believe that there is still room for improvement. We first focus on some

possible future works for *program splicing*. SPLICER relies on code search and enumerative search, and future works can focus on improving these two parts. To improve the code search component, one can switch to use more advanced natural language model such as `Word2Vec` [42], `FastText` [69] and recently developed `BERT` [70] to enhance the accuracy of code search using natural language. The training data could be the text specific to programming, similar to our method described in chapter 3. Using web data or relying on a popular search engine has also been proved to help code search [34, 57]. Programmers frequently use an online search engine to search for code snippets. Websites such as `StackOverflow` [62] have enormous program snippets associated with text which could be used to train statistical models.

Another future work direction for SPLICER could focus on improving the search algorithm. The current synthesis technique is an enumerative search algorithm which uses types and roles as heuristics. The problem with this enumerative search is its low efficiency; the search space grows exponentially which prevents the algorithm from synthesizing large programs quickly. This is a common problem with search-based program synthesis. Traditional search-based synthesis algorithms typically rely on using additional hints such as types [30, 77], templates [8] and, syntax [7, 29] to mitigate the scaling problem and thus to generate practical results. These hints are similar to the heuristics we used in SPLICER. The problem with these heuristics is that they typically do not generalize to other problems. Statistical methods, in contrast, tend to be very general and can be applied to many problem domains. DeepCoder [23] has shown that idioms learned from a code corpus can be used to reduce the search space significantly and thus to speed up the synthesis search process. These idioms can generalize to other synthesis problems. In addition, this type of search space reduction which gained from a code corpus typically cannot be derived using the hints

and heuristics mentioned above. Besides statistical methods, applying better search algorithm such as Monte Carlo tree search [99] (MCTS) could be another direction. The benefit of MCTS is its flexibility; one can use smart sampling techniques which may use deep learning to have better reward estimates. However, one important cost of such flexibility is the rarity of the solution in the search space. The search space is so large that it is difficult to have meaningful reward estimations. One might need to develop a smart estimation to overcome this problem.

Having discussed the potential future works for *program splicing*, let us discuss the possible direction to improve our API refactoring algorithm. As we described earlier, our API refactoring algorithm depends on first translating the API calls and then synthesizing a complete API call sequence. Naturally, two directions for improvement focus on API translation and API call sequence synthesis respectively. The problem of our API translation is that it completely relies on calculating the similarities of some specific natural language features. For libraries where the terminologies are similar, this works quite well with a standard language model such as `FastText` [69] and `BERT` [70]. However, as our evaluation indicated, this fails to work if two libraries use different terminologies. The reason is that the terminologies tend to have different semantic meaning, which prevents us from calculating semantic similarities. To overcome this issue, one could use a collection of similar API call sequence to train a statistical model for translation, as suggested in this paper [17]. This will eliminate the dependence on natural language features. In addition, using API call sequence pairs allows models to learn one-to-many API translations, which might give higher translation accuracy.

Having discussed the potential improvements for the projects proposed in this thesis, we believed that many problems related to "big code" still remain to be solved.

Applying corpus-driven methods to develop programming systems is still in its infant stage. As more and more data starts to stack up and computing power gets stronger and stronger, researchers will be able to utilize more and more resources to develop advanced AI-powered programming systems. Even though machines still cannot write programs for us, we believe that someday those AI-powered programming systems will further simplify programming and debugging, improve the quality of source code and even generate programs automatically without clear specifications from humans. In summary, the potential of corpus-driven programming systems is enormous.

Bibliography

[1] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," *arXiv preprint arXiv:1703.05698*, 2017.

[2] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, (New York, NY, USA), pp. 111–124, ACM, 2015.

[3] "Junit," 2018. Accessed: 2019-03-09.

[4] "Jira," 2019. Accessed: 2019-03-30.

[5] "Git," 2019. Accessed: 2019-03-30.

[6] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[7] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 404–415, ACM, 2006.

[8] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 497–518, 2012.

[9] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 317–330, ACM, 2011.

[10] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 50–61, ACM, 2011.

[11] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, (New York, NY, USA), pp. 13–24, ACM, 2010.

[12] B. Boehm, "The future of software and systems engineering processes," *University of Southern California, Los Angeles, CA*, pp. 90089–0781, 2005.

[13] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *ACM SIGPLAN Notices*, vol. 40, pp. 265–279, ACM, 2005.

[14] R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams, "Discovering likely mappings between apis using text mining," in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pp. 231–240, IEEE, 2015.

[15] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pp. 756–758, IEEE, 2016.

[16] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, pp. 325–334, IEEE, 2010.

[17] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between apis," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 82–91, IEEE Press, 2013.

[18] A. Gokhale, D. Kim, and V. Ganapathy, "Data-driven inference of api mappings," in *Proceedings of the 2nd Workshop on Programming for Mobile & Touch*, pp. 29–32, ACM, 2014.

[19] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 457–468, ACM, 2014.

[20] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based Semantic Code Search over Partial Programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 997–1016, ACM, 2012.

[21] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 419–428, ACM, 2014.

[22] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian specification learning for

finding api usage errors," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 151–162, ACM, 2017.

[23] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[24] Z. Manna and R. Waldinger, "Fundamentals of deductive program synthesis," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 674–704, Aug. 1992.

[25] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pp. 1–8, IEEE, Oct. 2013.

[26] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 166–178, ACM, 2015.

[27] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), pp. 43–54, ACM, 2015.

[28] A. Pnueli and R. Rosner, "On the synthesis of an asynchronous reactive module," in *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, ICALP '89, (London, UK, UK), pp. 652–671, Springer-Verlag, 1989.

[29] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia,

R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," *Dependable Software Systems Engineering*, vol. 40, pp. 1–25, 2015.

[30] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), pp. 229–239, ACM, 2015.

[31] T. Mitchell, B. Buchanan, G. DeJong, T. Dietterich, P. Rosenbloom, and A. Waibel, "Machine learning," *Annual review of computer science*, vol. 4, no. 1, pp. 417–433, 1990.

[32] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, (New York, NY, USA), pp. 257–269, ACM, 2015.

[33] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *ACM SIGPLAN Notices*, vol. 50, pp. 43–54, ACM, 2015.

[34] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean," *arXiv preprint arXiv:1511.08497*, 2015.

[35] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pp. 83–92, IEEE, 2004.

[36] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 485–495, IEEE Computer Society, 2009.

[37] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri, "Synthesizing transformations on hierarchically structured data," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 508–521, ACM, 2016.

[38] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *ACM SIGPLAN Notices*, vol. 51, pp. 761–774, ACM, 2016.

[39] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847, IEEE, 2012.

[40] V. Kashyap, D. B. Brown, B. Liblit, D. Melski, and T. W. Reps, "Source forager: A search engine for similar source code," *CoRR*, vol. abs/1706.02769, 2017.

[41] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 307–318, IEEE Computer Society, 2009.

[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[43] H. Sajnani, V. Saini, J. Ossher, and C. Lopes, "Is popularity a measure of quality? an analysis of maven components," in *Software Maintenance and Evolution*

(ICSME), 2014 IEEE International Conference on, pp. 231–240, Sept 2014.

[44] J. Ossher, H. Sajnani, and C. Lopes, "Astra: Bottom-up construction of structured artifact repositories," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 41–50, 2012.

[45] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241 – 259, 2014.

[46] A. Solar-Lezama, "The sketching approach to program synthesis," in *Asian Symposium on Programming Languages and Systems*, pp. 4–13, Springer, 2009.

[47] H. Feild, D. Binkley, and D. Lawrie, "An empirical comparison of techniques for extracting concept abbreviations from identifiers," in *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06)*, Citeseer, 2006.

[48] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), pp. 599–612, ACM, 2017.

[49] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," *CoRR*, vol. abs/1611.07502, 2016.

[50] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 107–126, 2015.

[51] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, 2013.

[52] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," *Dependable Software Systems Engineering*, vol. 40, pp. 1–25, 2015.

[53] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 243–253, IEEE Computer Society, 2009.

[54] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE software*, vol. 25, no. 5, 2008.

[55] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "Applying test-driven code search to the reuse of auxiliary functionality," in *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 476–482, ACM, 2009.

[56] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss, "Hunter: Next-generation code reuse for java," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 1028–1032, ACM, 2016.

[57] T. Gvero and V. Kuncak, "Interactive synthesis using free-form queries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 689–692, IEEE, 2015.

[58] "Beanshell," 2017. Accessed: 2017-04-04.

[59] "Nailgun," 2017. Accessed: 2017-04-04.

[60] B. Efron, *The jackknife, the bootstrap and other resampling plans*. SIAM, 1982.

[61] "Jsoup," 2017. Accessed: 2017-04-02.

[62] "Stackoverflow," 2017. Accessed: 2017-04-02.

[63] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 195–204, ACM, 2010.

[64] "Project grizzly," 2018. Accessed: 2018-09-04.

[65] "Apache commons net," 2018. Accessed: 2018-09-04.

[66] "Sshj," 2018. Accessed: 2018-09-04.

[67] "Itextpdf," 2018. Accessed: 2018-09-04.

[68] "Pdfbox," 2018. Accessed: 2018-09-04.

[69] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[70] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[71] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski, "Program splicing," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 338–349, ACM, 2018.

[72] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, (Berlin, Heidelberg), pp. 226–238, Springer-Verlag, 2005.

[73] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 530–541, ACM, 2014.

[74] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, (New York, NY, USA), pp. 316–329, ACM, 2010.

[75] P. D. Summers, "A methodology for lisp program construction from examples," *J. ACM*, vol. 24, pp. 161–175, Jan. 1977.

[76] E. Kitzelmann and U. Schmid, "Inductive synthesis of functional programs: An explanation based generalization approach," *J. Mach. Learn. Res.*, vol. 7, pp. 429–454, Dec. 2006.

[77] P.-M. Osera and S. Zdancewic, "Type-and-example-directed program synthesis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), pp. 619–630, ACM, 2015.

[78] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, pp. 20:1–20:44, Feb. 2013.

[79] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 630–640, IEEE, 2015.

[80] M. Harman, Y. Jia, and W. B. Langdon, "Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system," in *International Symposium on Search Based Software Engineering*, pp. 247–252, Springer, 2014.

[81] A. Marginean, E. T. Barr, M. Harman, and Y. Jia, "Automated transplantation of call graph and layout features into kate," in *International Symposium on Search Based Software Engineering*, pp. 262–268, Springer, 2015.

[82] Y. Jia, M. Harman, W. B. Langdon, and A. Marginean, "Grow and serve: Growing django citation services using sbse," in *International Symposium on Search Based Software Engineering*, pp. 269–275, Springer, 2015.

[83] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a c++ program to a problem class," in *European Conference on Genetic Programming*, pp. 137–149, Springer, 2014.

[84] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner, "Learning to Grade Student Programs in a Massive Open Online Course," in *Data Mining (ICDM), 2014 IEEE International Conference on*, pp. 785–790, IEEE, Dec. 2014.

[85] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model

for code," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 858–868, IEEE Press, 2015.

[86] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, 2016.

[87] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[88] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105, IEEE Computer Society, 2007.

[89] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 664–675, ACM, 2014.

[90] K. T. Stolee and S. Elbaum, "Toward semantic search via smt solver," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 25, ACM, 2012.

[91] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 295–306, IEEE, 2015.

[92] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Notices*, vol. 49, pp. 349–360, ACM, 2014.

[93] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ACM/IEEE International Conference on Software Engineering*, (New York, New York, USA), ACM Press, May 2010.

[94] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *International conference on Software engineering*, (New York, New York, USA), ACM Press, May 2011.

[95] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pp. 289–298, 2012.

[96] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pp. 47–50, IEEE, 2017.

[97] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 383–392, ACM, 2009.

[98] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *ACM Sigplan Notices*, vol. 45, pp. 302–321, ACM, 2010.

[99] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte

carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.