Program Splicing

Yanxin Lu
Rice Universty
Houston, Texas
yanxin.lu@rice.edu

Swarat Chaudhuri
Rice University
Houston, Texas
swarat@rice.edu

Chris Jermaine
Rice University
Houston, Texas
cmj4@rice.edu

David Melski
Grammatech, Inc.
Ithaca, NY
melski@grammatech.com

ABSTRACT

We 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. The synthesis process happens in two stages. First, the synthesizer identifies a small number of programs in the database that are relevant to the synthesis task. Next it uses an enumerative search to systematically fill the draft with expressions and statements from these relevant programs. The resulting program is returned to the programmer, who can modify it and possibly invoke additional rounds of synthesis.

We present an implementation of program splicing, 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 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.

ACM Reference Format:
Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2018. Program Splicing. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 12 pages.
https://doi.org/10.1145/3180155.3180190

1 INTRODUCTION

Copying and pasting from existing code is a coding practice that refuses to die out in spite of much expert disapproval [24, 28]. 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 this paper, 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 [5, 48] 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.

The primary distinction between our synthesis algorithm and existing approaches to synthesis lies in the use of pre-existing code. A key benefit of our data-driven approach is that it helps with the

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180190

```
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 1: Reading from CSV files

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 program splicing, called Splicer, that uses a corpus of approximately 3.5 million methods, extracted from the Sourcerer [7, 35, 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 [9], 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 the paper:

- 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 the paper is organized as follows. In Section 2, we give an overview of our method. Section 3 states our synthesis problem; Section 4 describes the approach of program splicing; Section 5 presents our evaluation. Related work is described in Section 6. We conclude with some discussion in Section 7.

2 OVERVIEW

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

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.

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 [47, 48] (Figure 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 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

```
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: Reading CSV: draft for matrix multiplication

```
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 < N; ++i) {
    String line = scanner.nextLine();
    String[] fields = line.split(",");
    for(int j = 0; j < N; ++j)
    mat[i][j]=Integer.parseInt(fields[j]);}
  int[][] mat2 = new int[N][N];
  for(int i = 0; i < N; ++i)
    for(int j = 0, s = 0; j < N; ++j) {
      for(int k = 0; k < N; ++k) s+= mat[i][k]*mat[k][j];
      mat2[i][j] = s;}
  return mat2;}
```

Figure 3: Reading CSV: Complete Program

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 "REQ" is run in an independent environment which has no access to the variables and names in the draft program. For example, in Figure 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 implementation is shown in Figure 1c). The system now extracts a set of *codelets* — expressions and statements — from these functions, and uses a

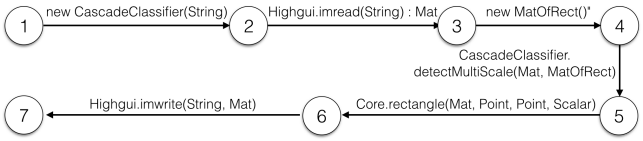

Figure 4: API call sequence constraint for face detection

composition of these codelets to fill in the hole in the draft. The completed draft is showed in Figure 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. Splicer now searches the code database for snippets that perform matrix 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 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 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 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 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 keep track of the program state which is checked against the requirement. For example in Figure 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`.

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

```
/* COMMENT: Doing face detection using OpenCV
 * REQ: API_cons("FaceDetectionTest.java");
 * __solution__
 * run_and_test(_has_detector_ && _has_image_ &&
 *              _has_detection_ && _image_written_); */
public void run() {
  String input_img = "lena.jpg";
  String output_img = "faceDetection.png";
  CascadeClassifier detector =new CascadeClassifier(??);
  ??}
```

Figure 5: Face Detection: Draft Program

```
public void run() {
  String input_image = "lena.png";
  String filename = "faceDetection.png";
  CascadeClassifier detector =
    new CascadeClassifier(getClass().getResource("
     lbpcascade_frontalface.xml").getPath());
  Mat image = Highgui.imread(getClass().getResource(
     input_image).getPath());
  MatOfRect faceDetections = new MatOfRect();
  detector.detectMultiScale(image, faceDetections);
  for (Rect rect : faceDetections.toArray()) {
    Core.rectangle(image, new Point(rect.x, rect.y), new
     Point(rect.x + rect.width, rect.y + rect.height),
     new Scalar(0, 255, 0));
  }
  Highgui.imwrite(filename, image);}}
```

Figure 6: Face Detection: Complete Program

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 6.

Note that the external code information in this example is provided above the function, as shown in Figure 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.

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.

$\langle expr \rangle ::= id \mid c \mid \langle expr \rangle\ binop\ \langle expr \rangle \mid unaryop\ \langle expr \rangle$
$\quad \mid \quad f(\langle expr \rangle, \ldots, \langle expr \rangle) \mid id := \langle expr \rangle \mid ??$

$\langle stmt \rangle ::= \textbf{let}\ id = \langle expr \rangle \mid \textbf{if}\ \langle expr \rangle\ \langle stmt \rangle\ \langle stmt \rangle$
$\quad \mid \quad \textbf{while}\ \langle expr \rangle\ \langle stmt \rangle \mid \langle stmt \rangle\ ;\ \langle stmt \rangle \mid ??$

$\langle program \rangle ::= id\ (\langle expr \rangle, \ldots, \langle expr \rangle)\ \langle stmt \rangle$

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 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 C. Let $U : \mathcal{L} \rightarrow \{True, False\}$ 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 $U(P_c^*) = True$.

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.

4.1 Searching for programs

In this section, we describe the code search techniques from Kashyap et al. [25] 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 [12] 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 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 [25] 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.

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 [5, 13–15, 37, 38, 51, 53], 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:

4.2.1 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 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

Procedure 1 fill

Input: A draft program, $P_s \in \mathcal{L}$ and a database program $P_d \in \mathcal{L}$
Output: A complete program P_c

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

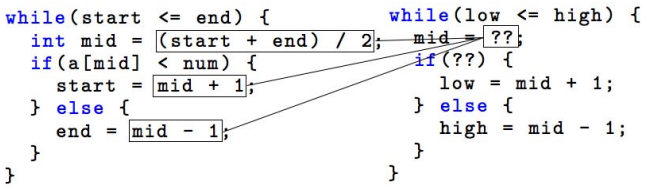

Figure 7: Matching for expression codelet

Procedure 2 merge

Input: A completed draft program, $P_s \in \mathcal{L}$ and a database program $P_d \in \mathcal{L}$
Output: A correct completion P_c

1: **if** no_undefined_refs(P_s) **then**
2: **if** is_correct(P_s) **then return** P_s **end if**
3: **return** *null*
4: **end if**
5: **for** $u \leftarrow$ next_undefined_ref(U), $r \leftarrow$ next_ref(P_d) **do**
6: **if** same_type(P_s, u, r) **then**
7: $P'_s \leftarrow$ substitute(P_s, u, r)
8: $P_c \leftarrow$ merge(P'_s, P_d)
9: **if** $P_c \neq null$ **then return** P_c **end if**
10: **end if**
11: **end for**
12: **return** *null*

to be the `rval` of other assignment statements. We can then 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.

4.2.2 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 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 [9].

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.

4.2.3 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.

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.

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 [48], syntax-guided synthesis [6], code reuse tools [21, 29, 43, 52] or other statistical methods [17, 39]. 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 [39] and anyCode [17] 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 [9] 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 8 shows the draft program.

5.2 Experiments

We implemented program splicing in Scala 2.12.1 based on 64-bit OpenJDK 8 and we used BeanShell [1] and Nailgun [3] 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

```
/* COMMENT: Setting up an HTTP server that serves the
 *          content of a local file
 * REQ:
 * import com.sun.net.httpserver.*, java.io.OutputStream;
 * __solution__
 * HttpServer server = http("http_test.txt", 23456);
 * test_server(new URL("http://localhost:23456/"));
 * server.stop(0); */
public HttpServer http(String filename, int port) {
  String content;
  // read the content of the file
  ??
  HttpServer server;
  HttpHandler handler = new HttpHandler() {
    public void handle(HttpExchange he) {
      he.sendResponseHeaders(??, content.length());
      OutputStream os = he.getResponseBody();
      os.write(content.getBytes()); os.close();
    }};
  // set up an http server
  ??
  return server;}

```

Figure 8: Setting up an HTTP server that serves the content of a text file

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 [7, 35, 45]. We extracted over 3.5 million methods with features from this corpus.

5.2.1 Synthesis Algorithm Evaluation. Table 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 4.1.

According the results showed in Table 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 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.

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

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 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 [9] 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 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.

5.2.2 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

```
/** TODO 1: Use Sieve of Eratosthenes to test primality
 *          of the given integer. */
static boolean sieve(int n) {
  boolean[] primes = new boolean[100];
  return primes[n];}

/** TODO 2: Test the sieve of Eratosthenes you've just
 *  written. Make sure to test the program with the
 *  following inputs: n = {1, 2, 3, ..., 73}
 *  Return true if the program is correct. */
static public boolean test() {return false;}

```

Figure 9: `Sieve of Eratosthenes` Skeleton with Tests

(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 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 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:

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

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[11] 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.

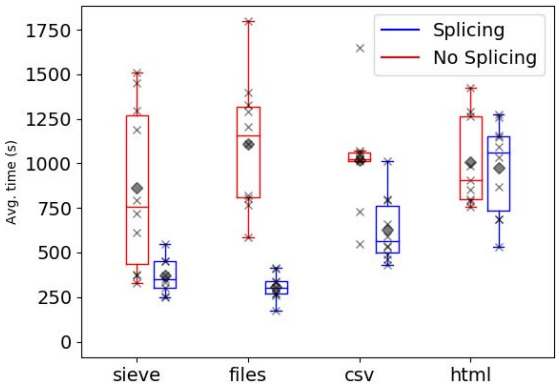

Figure 10: Time spent on each programming problem, with and without splicing.

`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 [2] 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 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 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 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` [2], 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

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

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 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 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 [4], 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.

5.2.3 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.

6 RELATED WORK

The notion of drafts used in program splicing is inspired by [48, 49] from the program synthesis field [5, 13–15, 37, 38, 51, 53]. However, the key difference is that our method reuses existing source code from the web instead of generating programs from scratch. Gilligan [20] and other genetic programming methods [9, 18, 22, 30, 36] are very similar to our work. However, they do not considering searching a large code corpus and suffer from efficiency concerns. CodePhage [46] transplants arbitrary code snippets across different applications, but the transplantaions are done exclusively for binary programs. Code reuse tools [21, 29, 43, 52] usually consider 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.

Statistical methods [8, 19, 32, 34, 40–42] have been shown to be quite effective in inferring program properties and code completion. Our work, however, does not depend heavily on statistical methods. Web search combined with natural language queries are used for code search [17, 39], but they do not consider draft programs that offer a context with which new programs must be merged. The problem of copy-and-paste is considered in [33], but only in the context of merging similar programs.

Our code search technique is explained extensively in [25]. Much of the related work on finding similar code has focused on *clone detection*: finding syntactically exact or nearly-exact copies of source code fragments. See [44] for a relatively recent survey of these techniques. Syntax elements have been shown to be effective in code search [23, 27], but our code search mainly relies on natural language with minor syntax elements. Code search has been performed in various other settings, using different code features [10, 16, 31]. Refer to Table 3 in [31] for a comprehensive comparison of various code search engines: most of these are targeted towards creating user-facing code search engines that can find relevant code based on a user-specified query. SMT solvers and semantic code search [26, 50] have also been used for semantic code search and program repair.

7 CONCLUSION

In this paper, 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.

One important piece of future work is to ensure the high quality of the database programs, because the effect of code quality on the quality of the synthesized code could be very significant. We could develop better features and similarity metrics to increase the code search precision.

REFERENCES

[1] 2017. BeanShell. (2017). https://github.com/beanshell/beanshell Accessed: 2017-04-04.
[2] 2017. JSoup. (2017). https://jsoup.org Accessed: 2017-04-02.
[3] 2017. Nailgun. (2017). http://martiansoftware.com/nailgun/ Accessed: 2017-04-04.
[4] 2017. Stackoverflow. (2017). https://stackoverflow.com Accessed: 2017-04-02.

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

[6] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

[7] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. 2014. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming* 79 (2014), 241 – 259. DOI: http://dx.doi.org/10.1016/j.scico.2012.04.008

[8] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *arXiv preprint arXiv:1611.01989* (2016).

[9] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 257–269. DOI: http://dx.doi.org/10.1145/2771783.2771796

[10] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 349–360.

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

[12] Henry Feild, David Binkley, and Dawn Lawrie. 2006. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEAâĂŹ06)*. Citeseer.

[13] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2016. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. *CoRR* abs/1611.07502 (2016). http://arxiv.org/abs/1611.07502

[14] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 599–612. DOI: http://dx.doi.org/10.1145/3009837.3009851

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 229–239. DOI: http://dx.doi.org/10.1145/2737924.2737977

[16] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. 2010. A search engine for finding highly relevant applications. In *ACM/IEEE International Conference on Software Engineering*. ACM Press, New York, New York, USA.

[17] Tihomir Gvero and Viktor Kuncak. 2015. Interactive synthesis using free-form queries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 689–692.

[18] Mark Harman, Yue Jia, and William B Langdon. 2014. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In *International Symposium on Search Based Software Engineering*. Springer, 247–252.

[19] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[20] Reid Holmes and Robert J. Walker. 2013. Systematizing Pragmatic Software Reuse. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 20 (Feb. 2013), 44 pages. DOI: http://dx.doi.org/10.1145/2377656.2377657

[21] Oliver Hummel, Werner Janjic, and Colin Atkinson. 2008. Code conjurer: Pulling reusable software out of thin air. *IEEE software* 25, 5 (2008).

[22] Yue Jia, Mark Harman, William B Langdon, and Alexandru Marginean. 2015. Grow and serve: Growing Django citation services using SBSE. In *International Symposium on Search Based Software Engineering*. Springer, 269–275.

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[24] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 485–495.

[25] Vineeth Kashyap, David Bingham Brown, Ben Liblit, David Melski, and Thomas W. Reps. 2017. Source Forager: A Search Engine for Similar Source Code. *CoRR* abs/1706.02769 (2017). http://arxiv.org/abs/1706.02769

[26] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 295–306.

[27] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[28] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. 2004. An ethnographic study of copy and paste programming practices in OOPL. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 83–92.

[29] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. 2009. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 476–482.

[30] Alexandru Marginean, Earl T Barr, Mark Harman, and Yue Jia. 2015. Automated transplantation of call graph and layout features into Kate. In *International Symposium on Search Based Software Engineering*. Springer, 262–268.

[31] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *International conference on Software engineering*. ACM Press, New York, New York, USA.

[32] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. 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)*. ACM, New York, NY, USA, 997–1016. DOI: http://dx.doi.org/10.1145/2384616.2384689

[33] Krishna Narasimhan and Christoph Reichenbach. 2015. Copy and Paste Redeemed. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 630–640.

[34] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 858–868.

[35] J. Ossher, H. Sajnani, and C. Lopes. 2012. Astra: Bottom-up Construction of Structured Artifact Repositories. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. 41–50. DOI: http://dx.doi.org/10.1109/WCRE.2012.14

[36] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. 2014. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming*. Springer, 137–149.

[37] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of an Asynchronous Reactive Module. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP '89)*. Springer-Verlag, London, UK, UK, 652–671. http://dl.acm.org/citation.cfm?id=646243.681607

[38] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

[39] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2015. SWIM: Synthesizing What I Mean. *arXiv preprint arXiv:1511.08497* (2015).

[40] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 761–774.

[41] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. DOI: http://dx.doi.org/10.1145/2676726.2677009

[42] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. DOI: http://dx.doi.org/10.1145/2594291.2594321

[43] Steven P. Reiss. 2009. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 243–253. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070525

[44] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[45] H. Sajnani, V. Saini, J. Ossher, and C.V. Lopes. 2014. Is Popularity a Measure of Quality? An Analysis of Maven Components. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. 231–240. DOI: http://dx.doi.org/10.1109/ICSME.2014.45

[46] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 43–54.

[47] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*. Springer, 4–13.

[48] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. DOI: http://dx.doi.org/10.1145/1168857.1168907

[49] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2012. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5 (2012), 497–518. DOI: http://dx.doi.org/10.1007/s10009-012-0223-4

[50] Kathryn T Stolee and Sebastian Elbaum. 2012. Toward semantic search via SMT solver. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 25.

[51] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[52] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. 2016. Hunter: Next-generation Code Reuse for Java. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 1028–1032. DOI: http://dx.doi.org/10.1145/2950290.2983934

[53] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 508–521. DOI: http://dx.doi.org/10.1145/2908080.2908088