Using Web Corpus Statistics for Program Analysis

Chun-Hung Hsiao Michael Cafarella Satish Narayanasamy
University of Michigan
{chhsiao,michjc,nsatish}@umich.edu

Abstract

Several program analysis tools—such as plagiarism detection and bug finding—rely on knowing a piece of code's relative semantic importance. For example, a plagiarism detector should not bother reporting two programs that have an identical simple loop counter test, but should report programs that share more distinctive code. Traditional program analysis techniques (*e.g.*, finding data and control dependencies) are useful, but do not say how surprising or common a line of code is. Natural language processing researchers have encountered a similar problem and addressed it using an n-gram model of text frequency, derived from statistics computed over text corpora.

We propose and compute an n-gram model for programming languages, computed over a corpus of 2.8 million JavaScript programs we downloaded from the Web. In contrast to previous techniques, we describe a code n-gram as a subgraph of the program dependence graph that contains all nodes and edges reachable in n steps from the statement. We can count n-grams in a program and count the frequency of n-grams in the corpus, enabling us to compute *tf-idf*-style measures that capture the differing importance of different lines of code. We demonstrate the power of this approach by implementing a plagiarism detector with accuracy that beats previous techniques, and a bug-finding tool that discovered over a dozen previously unknown bugs in a collection of real deployed programs.

1. Introduction

Standard data flow and control flow analysis methods form the basis of many programmer productivity tools. However, these methods have been largely limited to analyzing one program or a set of programs chosen by the programmer. Today, one could easily collect source code for millions of programs by crawling the Web, especially for web applications written in scripting languages like JavaScript. Programming assignments submitted by thousands of students enrolled on massive open online courses (MOOCs) are another rich source of programs. By examining these large corpora of source code and computing the probability of observing programming patterns, we aim to build a statistical body of knowledge that can improve a wide range of program analysis.

The natural language processing (NLP) community has discovered a technique that is broadly useful to many individual language tasks: exploiting usage statistics computed over large corpora of text. Simple counts over observed words and sequences can be useful in information retrieval [26], machine translation [2], spelling correction [9], and even analysis of historical texts [19, 23].

We postulate that a similar corpus-driven "big-code" approach can work for program analysis. That is, the information that we can glean from analyzing millions of unrelated programs could allow us to better analyze a new program. In this paper, we focus on exploiting one particular information that we can gather by analyzing a large corpus: "importance" of a code snippet. We consider a code snippet that is seen in many thousands of programs to be less important than one that is rarely seen. We demonstrate the utility of this information by using it to improve the accuracy of two tools: a plagiarism checker and a copy-paste bug finder. We have built these tools for JavaScript programs, because a corpus-driven program analysis is particularly attractive for such programs. Hundreds of thousands of web applications developed today are written in JavaScript and are easily available to build a large corpus.

The first step that we take to realize the above goals is to develop a method to extract meaningful code snippets ("semantic tokens") from the raw source code of millions of unrelated programs such that they are amenable for statistical analysis. The standard textual *n-gram* statistical method used in NLP applications relies on certain properties of natural language text: the context of a token is reasonably captured by the preceding words, and the text tokens are different enough to have distinctive distributions, but common enough that a single text token can be observed

OOPSLA '14, October 20–24, 2014, Portland, Oregon, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

multiple times. Source code does not obviously have these features: programs have complicated dependencies among non-sequential tokens, some programming language tokens are so common (*e.g.*, "int" or an open brace) that it is unclear what information they carry, and other tokens (variable names) are so idiosyncratic that it is unclear how to count them across multiple programs. Thus, an important contribution of this paper lies in building useful "semantic tokens" from the raw JavaScript programs; these are small abstracted versions of program code under which the n-gram counting model can be successful.

We build a database of n-gram code snippets along with their importance scores. We use this information to build two tools: a plagiarism checker and a copy-paste bug finder. A traditional plagiarism detection tool might identify two regions of code that are quite similar, but in fact reflect a repetitive and dull task, such as a simple increment-and-test loop. A tool with corpus-derived knowledge of the importance of code snippets would recognize that overlap on such common tasks is poor evidence of plagiarism.

We also build a copy-paste bug finder tool that uses our plagiarism checker to find code snippets in a program that have been potentially copied from other sources, and then detects errors that programmers are likely to make while using the old code in the new context.

Background — The use of corpus statistics in information retrieval and natural language processing is well known [2, 26]. Statistical debugging is an active area of research [15, 17], but to date has focused on statistical analysis of program traces rather than source code. Existing methods for our two target applications (plagiarism detection [4, 6, 7, 10, 11, 13, 16, 22, 25] and copy-paste bug finding [3, 11, 14]) do not exploit corpus-derived data. While past work has used sequential n-grams to characterize programs [7, 12], none has considered n-grams based on program dependence graphs [1].

Contributions and Outline — The central contributions of this work include the following:

- A model for extending corpus-driven statistics approaches to programmatic tasks (Section 2).
- Techniques for addressing a basic challenge when using statistical methods—identifying relevant semantic tokens. We describe algorithms for applying these techniques to a corpus of 2.8 million JavaScript programs downloaded from the Web (Section 3).
- A demonstration of the effectiveness of corpus-driven code statistics on two automated programming tasks: plagiarism detection (Section 4) and copy-paste bug finding (Section 5). In the former case, our corpus-driven approach beats a known baseline. In the latter, our method has found over a dozen previously unknown copy-paste bugs (Section 6). Also we present a first tool to find copy-paste bugs in JavaScript.

We discuss related work in Section 7. In Section 8 we conclude with a discussion of future work.

2. N-Gram Data Model

In this section, we describe our model for computing corpus statistics for programming languages over a collection of source code. This model includes two elements: (1) a definition of "semantic tokens" that can capture small-grained topics of source code while being general enough that identical tokens appear in multiple programs; and (2) a statistical method for computing the importance of such a token in a program. The results in this paper focus on processing JavaScript, and we will use the code example below to explain our system. However, our model has very little that is language-specific: it should apply to any imperative-style programming language, including C, C++, Python, and Java.

To explain the two elements of the model, we use the following JavaScript subroutine as a running example. It takes a value and an array as parameters, then returns true only if the array contains the given value.

```
function inArray(a, val) {
  var i;
  for (i = 0; i < a.length; i++) {
    if (a[i] === val) {
      return true;
    }
  }
  return false;
}
```

2.1 Programmatic N-Grams

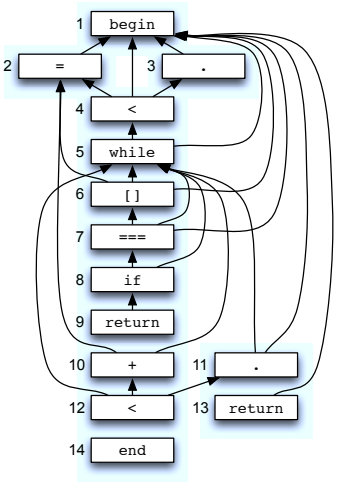
Some assumptions behind standard linguistic n-gram[1] construction are so straightforward as to be barely noticeable: tokens are delimited by whitespace or sentence boundaries, and the context of a token is well-captured by its preceding text. Of course these qualities are only rough approximations of the truth. Collocations (*e.g.*, *Procter and Gamble*) and words with unusual punctuation (*e.g.*, *I.B.M.*) violate the standard tokenization approach and often need special handling in text processing. Also, not all linguistic phenomena can be captured by simple short word sequences: some tasks require linguistic parse trees to describe long-range linkages between words. However, in general these qualities hold true often enough that the n-gram model works.

These qualities generally do not hold true for source code. In order to apply the n-gram approach to program source code, we must choose how to delimit a "gram," and how to decide when one gram immediately precedes another.

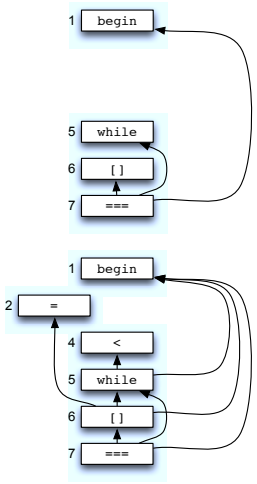
[1] An NLP researcher may object to this paper's use of "n-gram", which in some projects refers strictly to a multiword text probability model. However, widespread use of Google's "Ngram data," which comprises simple word counts, has undermined that strict meaning. For us, the term "n-gram" refers to sequential multitoken text strings, which are used to construct a statistical resource. We will treat *token* and 1-gram as interchangeable terms.

```
   function inArray(a, val) {
 1   begin;
 2   i = 0;
 3   $0 = a.length;
 4   $1 = i < $0;
 5   while ($1) {
 6     $2 = a[i];
 7     $3 = $2 === val;
 8     if ($3) {
 9       return true;
       }
10     i = i + 1;
11     $4 = a.length;
12     $1 = i < $4;
     }
13   return false;
14   end;
   }
```

(a)



(b)



(c)

Figure 1: (a) The canonical form of the `inArray` subroutine. (b) The program dependence graph of the `inArray` subroutine. A direct edge from s to t means that s is control- or data-dependent on t. (c) Two n-gram graphs. The upper graph is the 2-gram of line 7; the lower graph is the 3-gram of the same line.

2.1.1 Delimiting Tokens

There are at least two obvious choices when describing programmatic n-grams. The first is to model each gram as a single line of code. This appears natural to a human programmer, and program complexity is often described in pure line counts. However, the amount of program complexity carried by a single line can vary tremendously from line to line: contrast a simple integer increment with a deeply-nested set of mathematical operators that combine half a dozen variables. In the latter case, a simple mathematical operation might mean very different things depending on the nesting; however, a gram-per-line approach would ignore the nesting and treat all occurrences of the operation identically. Further, this approach is very sensitive to variable naming.

A second choice is to model each gram as a programming language token: `if`, `for`, variable names, and so on. However, low-level tokens carry very little information, requiring us to construct large consecutive grams to capture what appear to the programmer to be trivial steps. Also, consecutive pieces of source code may not have much to do with each other, as single programs often reorder different operations for application or performance reasons. To construct tokens with appropriate complexity, we propose to decompose each statement into a *canonical form* that is similar to *three address code*. Each statement in the source JavaScript yields at least one statement in canonical form, and often many more.

Three address code is an intermediate code representation often used in compilers and program optimization [20]. A single example of three address code includes a binary operator, its operands, and often an assignment (hence, there are at most three operands.) For example, the loop condition test from `inArray` is translated into a form that includes an equality test between two values and assignment of the result to a third location. Operands are internal symbolic names, not names from the program code.

Lines in our canonical form depend on each other to provide input values. Understanding the dependence structure is critical for understanding the original program. Not only are lines in the source language interrelated, but multiple lines of three address code can be derived from a single source line. We describe the dependence structure in detail below.

2.1.2 Ordering Tokens

NLP grams are generally built using simple linear lexical order instead of, say, an ordering imposed by a parse tree. Ordering of lines in source code is at times unimportant: independent statements can be arbitrarily reordered without changing the logic of the program. At other times—when there is a data or control dependency between statements—the order is crucial for code correctness and for understanding the programmer's goal.

Therefore, we compute grams not using lexical order of statements, but rather ordering imposed by the *program dependence graph* [5]. A node represents a single canonical form statement. A directed edge leads from node s to t when s takes as input a value written by t, or when s's execution depends on the result of t.

We show the program dependence graph of the `inArray` function in Figure 1b, with each node annotated by its corresponding canonical form line in Figure 1a. The initialization of loop variable `i` takes place at line 2. Because this piece of code is not dependent on anything except the invocation of the function itself, node 2 in Figure 1b points only to node 1, which represents `begin`. The loop bounds check at line 4 (from source code `i < a.length`) depends on both the value of `i` and the length of array `a`; hence, node 4 from Fig-

ure 1b points to both node 2 and node 3 (which represents the result of looking up `a.length`).

We will say that two canonical form lines are consecutive when there is an edge between their corresponding nodes in the graph. So line 4 from Figure 1a is *immediately preceded* by both line 2 and line 3. This model is quite different from the traditional linguistic n-gram construction, in which a given word in a text has at most one (obvious) predecessor.

Note that the program dependence graph in Figure 1b does not contain all control dependence edges. Details about how to construct the graph, including simplification rules to ensure that it is acyclic, are described in Section 3.2.

2.1.3 Constructing N-Grams

In computational linguistics, an n-gram consists of n consecutive tokens in a document. We construct an n-gram in a program similarly: starting from a particular token, we traverse the program dependence graph backward up to distance $(n-1)$, collecting all the visited grams and dependence edges during the traversal. In other words, an n-gram of a specific gram x is the subgraph of the program dependence graph consisting of all paths of length $(n-1)$ starting from x. Thus, the n-gram of x captures all recent dependent computations that x requires prior to its execution.

For example, the upper portion of Figure 1c is the 2-gram of line 7. It includes line 7 itself as well as lines 1, 5, and 6. The lower subgraph of Figure 1c is the 3-gram of line 7. It includes all nodes in the 2-gram as well as lines 2 and 4 (which are two steps away from line 7).

Because a single node (or, equivalently, statement) can have multiple immediate predecessors, the number n only tells the *depth* of an n-gram. The number of nodes in an n-gram also depends on its *width*, which is determined by the dependence graph. For example, the 2-gram of line 7 contains 4 nodes, whereas the 2-gram of line 9 contains only 2 nodes (lines 7 and 9). We describe n-gram construction in more detail in Section 3.3.

In summary, an n-gram in a program is a labeled subgraph of the program dependence graph constructed over the canonical form of the program's statements. The subgraph consists of all paths of length $(n-1)$ starting from a specific statement.

2.2 Quantifying Gram Importance

Our method to evaluate the "importance" of an n-gram is directly inspired by the *tf-idf* measure. There are several ways to define the term frequency (*tf*) of an n-gram x in a program P. One simple and well-known method is *boolean frequency* [18]:

$$tf(x,P) = \begin{cases} 1, & \text{if } x \in P, \\ 0, & \text{otherwise.} \end{cases}$$

This method seems appropriate to source code, where important code is determined more by the call graph than frequency in the source code text itself.

```
                                        2-gram   3-gram
                                        tf-idf   tf-idf
   function inArray(a, val) {
 1   begin;                             0.000    0.000
 2   i = 0;                             1.017    1.017
 3   $0 = a.length;                     0.969    0.969
 4   $1 = i < $0;                       2.238    2.876
 5   while ($1) {                       1.641    2.368
 6     $2 = a[i];                       3.035    3.590
 7     $3 = $2 === val;                 4.767    6.704
 8     if ($3) {                        4.560    5.296
 9       return true;                   1.699    5.911
       }
10     i = i + 1;                       1.934    2.232
11     $4 = a.length;                   2.024    2.312
12     $1 = i < $4;                     1.564    3.846
     }
13   return false;                      1.857    1.857
14   end;
   }
```

Figure 2: The 2-gram and 3-gram *tf-idf* scores for each canonical line of `inArray`.

Given a program corpus Π, the inverse document frequency (or *idf*) of an n-gram x measures the overall importance of each unique n-gram:

$$idf(x,\Pi) = \log \frac{|\Pi|}{|\{P \in \Pi : x \in P\}|}.$$

So the *tf-idf* measure for x in P with respect to corpus Π is the product of the two:

$$\textit{tf-idf}(x,P,\Pi) = tf(x,P) \times idf(x,\Pi).$$

If x is indeed in P, then the *tf-idf* measure of x is simply the same as its *idf* value.

Figure 2 shows the *tf-idf* score for each 2-gram and 3-gram in `inArray`. Of course, all grams here satisfy the boolean *tf* test, so *tf-idf* and *idf* are equivalent. For $n = 2$ and $n = 3$, lines 7–9 have the highest *tf-idf* values, and we might thus imagine that these lines are more remarkable or more important than the others in understanding the function. Indeed, to the programmer's eye this is obvious: lines 7–9 describe inner loop code that determines what the function will do in most cases. The remainder of `inArray` is just boilerplate loop iteration code.

This example also illustrates the potential power of larger values for n. Line 9 has a 2-gram of unremarkable *tf-idf* that reflects a fairly common programming motif: it tests a boolean, and if true, then the code returns true. However, its 3-gram has a comparatively quite high *tf-idf* that reflects a less common pattern: inside a `while` loop, the code tests two values for equality, and if they are equal, the code returns true. In practice, a user's specific choice for n will depend on the application and available data (as with text, at some point n is large enough that no large counts can be computed, because each observation is *sui generis*).

3. Computing Corpus Statistics

Computing n-gram statistics for text is relatively straightforward (even if very large text corpora and estimating smooth-

$$
\begin{aligned}
func &\rightarrow \texttt{function}\ id^{?}\texttt{(}var^{*}\texttt{)}\ \texttt{\{ begin;}\ stmt^{*}\ \texttt{end; \}}\\
stmt &\rightarrow assign\texttt{;} \mid \texttt{break;} \mid \texttt{continue;} \mid \texttt{return}\ val^{?}\texttt{;}\\
&\quad \mid \texttt{if (}\ val\ \texttt{) \{}\ stmt^{*}\ \texttt{\}}\ \texttt{else \{}\ stmt^{*}\ \texttt{\}}\\
&\quad \mid \texttt{while (}\ val\ \texttt{) \{}\ stmt^{*}\ \texttt{\}}\\
&\quad \mid \texttt{for (}\ var\ \texttt{in}\ val\ \texttt{) \{}\ stmt^{*}\ \texttt{\}}\\
&\quad \mid \texttt{switch (}\ val\ \texttt{) \{}\\
&\qquad (\texttt{case}\ val\texttt{:}\ stmt^{*})^{*}\ (\texttt{default:}\ stmt^{*})^{?}\texttt{\}}\\
&\quad \mid \texttt{with (}\ val\ \texttt{) \{}\ stmt^{*}\ \texttt{\}}\\
assign &\rightarrow var\ \texttt{=}\ val \mid var\ \texttt{=}\ op_{\text{unary}}\ val \mid var\ \texttt{=}\ val\ op_{\text{binary}}\ val\\
&\quad \mid var\ \texttt{=}\ val\ \texttt{?}\ val\ \texttt{:}\ val \mid var\ \texttt{=}\ val\texttt{.}identifier\\
&\quad \mid var\ \texttt{=}\ val\texttt{[}val\texttt{]} \mid var\ \texttt{=}\ identifier\texttt{(}val^{*}\texttt{)}\\
&\quad \mid var\ \texttt{=}\ \texttt{(}func\texttt{)(}val^{*}\texttt{)} \mid var\ \texttt{=}\ val\texttt{.}identifier\texttt{(}val^{*}\texttt{)}\\
&\quad \mid var\texttt{.}identifier\ \texttt{=}\ val \mid var\texttt{[}val\texttt{]}\ \texttt{=}\ val\\
val &\rightarrow var \mid literal \mid func
\end{aligned}
$$

Figure 3: The canonical form's formal definition. S^* means that S appears at 0 or more times, and $S^?$ means that S appears at most once.

ing statistics provide difficult challenges). However, translating these techniques to our gram model and corpus setting is not. We now show the technical challenge and algorithmic solution for each of the above-described steps: transforming JavaScript into canonical form, computing large numbers of program dependence graphs, and extracting many n-grams. Finally, we discuss how to count grams, a near-trivial step in the textual case that is challenging in ours.

3.1 Canonical JavaScript

Our goal in transforming a JavaScript program into canonical form is to decompose all statements — both complicated and simple ones — into units that carry a roughly standard amount of "semantic information." Figure 3 shows our design for the JavaScript canonical form, which is based on the three address code representation. We implemented our transformation by analyzing the abstract syntax tree (AST) produced by the V8 JavaScript engine. We accurately handled most language features in JavaScript. However, we do not handle a few features such as dynamic scoping and exception handling (Section A.5). Fortunately, our corpus-driven statistical approach can tolerate such small degrees of inaccuracy in our transformation.

In summary, each line in canonical form is one of:

- a handful of reserved operators: `while`, `if`, etc.
- a function invocation
- an assignment operation
- `begin` or `end` to represent function entry and exit.

Some high-level programming constructs, such as `if` and `while`, have representations in canonical form, with the restriction that their boolean test can only evaluate an input variable, not a complex expression. Some other constructs (such as `do...while`) are translated to an equivalent `while` statement.

Each assignment in canonical form contains:

- One operator, which can be unary, binary or ternary arithmetic; a field or array access; or a function or method invocation.
- Arguments in the form of variables or constants.
- A variable to store the result of the operation.

Rules for transforming a legal JavaScript program into canonical form appear in the Appendix (Section A). Our canonical form preserves certain pieces of program information, such as "which statements are in the same loop body" and "which statements are controlled by the same `if` condition." These are important facts about program structure that we want the n-grams to capture.

3.2 The Program Dependence Graph

Once we have transformed a JavaScript program into canonical form, we can build its *program dependence graph*. It is a directed graph $G = (V, E)$, where V is the set of all statements, and E contains all edges (s, t) such that the statement s is data-dependent or control-dependent on the statement t. The program dependence graph is a standard structure in compiler construction (and is covered in texts such as Aho, *et al.* [1]), but we have made a number of changes in order to fit the needs of extracting n-grams.

3.2.1 Data Dependence Edges

We perform a standard *reaching definition analysis* to identify all *use-def* chains between statements. The definition of a variable (that is, its assignment) at a statement t will *reach* its use at a statement s if s depends on the value assigned at t and there is no intervening definition of the variable between s and t. A use-def chain for a use of a variable is all reaching definitions of the use. For each use-def chain (t, s) where t is the definition and s is the use, the directed edge (s, t) is added into the program dependence graph, except:

1. When t is outside the function scope of s. Since the semantics of nested functions provides no control flow between t and s, it is incorrect to add such an edge. Instead, we model this dependency by assuming that the definition is implicitly passed into the function containing s, so we add edge (s, `begin`) to the program dependence graph, where `begin` is the begin statement of the function containing s.
2. When t appears after s in lexical order. This case can only appear in looping-style control flow constructs. Including this edge would make the program dependence graph more accurate since it preserves the inter-iteration dependency between s and t; but there would be many such edges in a loop and thus the graph would be much more

complex. We believe that the information these edges preserve is not worth the complexity they bring, so we exclude such edges in our design. We discuss our design decisions around cycles in more detail in Section 3.2.3.

In addition to the above, for each statement s that uses a parameter, and for each variable whose definition value is missing, we add $(s,$ `begin`$)$ to the graph. We essentially treat `begin` as a dummy statement that initializes the value of all parameters, variables in outer scopes, and all otherwise-uninitialized variables. In so doing, we ensure that all edges in the graph are *intra-procedural*.

3.2.2 Control Dependence Edges

Traditional optimizing compilers must construct edges to reflect control flow dependence as well as data dependence. In our case, we want to explicitly retain control flow structures such as `while` and `if` in our canonical version of three address form. Thus, instead of a standard control flow analysis we add control flow-inspired edges as follows:

1. Add $(s,$`begin`$)$ to the program dependence graph, where `begin` is the begin statement of the function that contains s. This edge is conventionally added only because of the ease of control flow analysis.
2. If s is inside the body of t, and t is one of the `if`, `while`, `for...in`, `switch` or `with` statements, then add (s,t) into the graph. That is, the execution of s depends on the execution of t.

We ignore all control dependence edges owing to `break`, `continue`, and `return` because they often lead to cycles in the program dependence graph.

3.2.3 Cycle Removal

An important design choice we made is to keep the program dependence graph *acyclic*. Because dependencies across different iterations of the same loop can only be expressed by cycles, we lose all information about inter-iteration dependencies. Keeping the cycles makes the program dependence graph more accurate, but also fills the related n-grams with large numbers of extra edges. Any gram found in a loop body would thus appear to be entirely different from the exact same gram found outside a loop. Distinguishing *in-loop* and *out-of-loop* versions of the same code might be useful in some cases, but we could not find any practical cases to warrant the extra complexity. Avoiding loops also makes it easier to compare and count n-grams, as we discuss below.

3.3 Extracting and Comparing N-Grams

We extract the set of all n-grams in a program according to Algorithm 1. For each statement in the program, we locate the corresponding node in the program dependence graph, and then perform a breadth-first search with a depth limit $(n-1)$. For each such search, we collect the visited nodes and edges to create the statement's n-gram.

Algorithm 1 The n-gram extraction algorithm.

```
function ExtractNGrams(n, P)
    P' ← χ(P)                                  ▷ Canonical form of P
    G ← PDG(P')                                ▷ Program dependence graph of P'
    Γ ← ∅                                      ▷ The set of all n-grams in P
    for p ∈ P' do
        Γ ← Γ ∪ {NGramBFS(G, p, n)}
    return Γ

function NGramBFS(G, v, n)
    V ← {v}                                    ▷ The set of vertices with distance ≤ n − 1
    E ← ∅                                      ▷ The set of edges with distance ≤ n − 1
    d[v] ← 0
    Q ← ∅
    Enqueue(Q, v)
    while Q ≠ ∅ do                             ▷ Breadth-first search with depth ≤ n − 1
        v ← Dequeue(Q)
        for (v, u) ∈ G do
            E ← E ∪ {(v, u)}
            if u ∉ V then
                V ← V ∪ {u}
                d[u] ← d[u] + 1
                if d[u] < n − 1 then
                    Enqueue(Q, u)
    return (V, E)
```

We apply Algorithm 1 to all programs in the downloaded JavaScript corpus to build the n-gram database. Collecting and counting this huge set of small graphs is an unusual challenge. Graph databases are a popular research area, but focus primarily on large graphs [31], on moderate numbers of graphs [28–30], or on frequent subgraph mining [27]. In contrast, we want to store and compute frequency information for a large number of small graphs; for our test corpus, approximately 662 million of them.

Counting graph frequencies is equivalent to computing many graph isomorphism problems. Doing so is in principle infeasible, but our graphs are generally very small and so an exhaustive approach is possible. We encode each n-gram graph as a string. For example, the 3-gram in Figure 1c consists of 6 nodes and is encoded as follows:

```
(begin)(<)(=)(while 0 1)([] 0 2 3)[=== 0 3 4]
```

First, we order all nodes, so the `begin` statement becomes node 0, the `<` operator becomes node 1, and so on. With this ordering, each node is encoded as a parenthesis-enclosed string that consists of the name of its operator followed by a list of the nodes it depends on. The "starting" node for the n-gram will not be depended upon by any other node; we denote this node with brackets rather than parentheses.

Since there are $k!$ different orderings for an n-gram of k nodes, there are many possible string representations for a given n-gram. To choose a single representation, we enumerate every ordering and find the lexically minimal string. Finding the minimal string for a given n-gram can be time-consuming, but k is generally small and we use a number of heuristics to avoid enumerating some orderings (*e.g.*, when we can detect that two orderings will yield the same string).

Once all n-grams are represented using these standardized strings, counting them is straightforward.

4. Plagiarism Detection

The first application for our corpus-driven technique is plagiarism detection. This system examines a set of programs to find pairs of code regions that have likely been copied from one location to the other. Such systems are useful when teaching classes that include programming tasks. The growth in online education, and the accompanying growth in students doing work outside traditional academic settings, will likely increase the need for systems that detect when students submit work that is not their own.

The problem of plagiarism detection has been extensively studied [16, 25]. However, all systems we know of are sensitive to the problem of "trivial plagiarism." The systems recognize similar code regions, but cannot detect when the similar region consists of trivial or widely-known code. For example, older JavaScript programs used a standard technique to detect the current browser engine; the repeated appearance of this idiom reflects common knowledge and practice, not rampant plagiarism. By using our n-gram corpus statistics, we can filter the output of a plagiarism detection system and remove plagiarism reports that describe uninteresting code regions. The result should be a hybrid plagiarism detector that reports many fewer false alarms.

4.1 Detection Algorithm

We assume the existence of a plagiarism detector that examines a set of programs and reports a set of suspicious program pairs. A good detector would report examples of plagiarism that are almost always true (that is, it has high precision) while also reporting almost all the plagiarism cases in the corpus (that is, it has high recall). By filtering the output of an existing plagiarism detection tool, we aim to improve its precision by removing false alarms that reflect "trivial plagiarism" of common idioms. Our filter should impact the core detector's recall as little as possible. It works by identifying commonly-used grams that should also have low *tf-idf* scores. Denote $\Gamma_\theta^{(n)}(X)$ as the set of n-grams of program X whose *tf-idf* scores are at least θ. For a suspicious program pair of plagiarism (P, Q), if $\Gamma_\theta^{(n)}(P) \cap \Gamma_\theta^{(n)}(Q) = \emptyset$, then we drop the pair from the output.

4.2 Discussion

Our filter is immune to plagiarism techniques that preserve control and data flow, but is vulnerable to certain changes in those flows. For example, consider a plagiarizer who generates a new plagiarized program by copying a source program, and then inserts `{ a++; a--; }` before every read of `a`. These new instructions modify the data flow observed at each read of `a`. All `a`-focused grams of size 2 or greater in the plagiarized program would appear to be quite different from their original version. If they also appear to be very common, the plagiarized program may be incorrectly removed by our filter. We could address this problem by searching for synthetic-seeming "nonsense" code that can be removed without impacting the code's output.

Note that our system can help detect code plagiarism, but is not designed for *content* plagiarism. Imagine two programs that consist entirely of calls to `document.write()`, which emit near-identical textual content. Assume the base plagiarism tool reports these as a suspicious pair; the grams' *tf-idf* scores will likely appear trivial and so will be removed by our filter. The problem, of course, is that the program *is* trivial, but the string content is not.

5. Copy-Paste Bug Finding

Our second application is a mechanism for finding *copy-paste bugs* in software. These arise when a user copies and pastes source code in order to create a new version that is similar but not identical; the programmer then fails to fully or correctly rename variables from the original version. Figure 4 illustrates such a bug. This problem can arise when the programmer copies her own code, or code found on the Internet. Such bugs are easy to introduce, but may be hard to find and diagnose. Other researchers have looked at copy-paste bugs [14].

```
 1 function FindParentLeft(Obj) {
 2   var curLeft = 0;
 3   if (Obj.offsetParent) {
 4     while (Obj && (null != Obj.offsetLeft)) {
 5        if (...) curLeft += Obj.offsetLeft;
 6        Obj = Obj.offsetParent;
 7     }
 8   } else if (Obj.x) {
 9     curLeft += Obj.x;
10   }
11   return (curLeft);
12 }

13 function FindParentTop(Obj) {
14   var curTop = 0;
15   if (Obj.offsetParent) {
16     while (Obj && (null != Obj.offsetTop)) {
17       if (...) curTop += Obj.offsetTop;
18       Obj = Obj.offsetParent;
19     }
20   } else if (Obj.x) {
21     curLeft += Obj.x;
22   }
23   return (curTop);
24 }
```

Figure 4: Two JavaScript functions downloaded from `http://www.petfoodindustry.com/WorkArea/java/webtoolbar.j` and slightly edited for clarity. The top `FindParentLeft` function was apparently incorrectly copied, pasted, and modified to create the bottom function `FindParentTop`. The programmer has failed to change `curLeft` to `curTop` at line 21. We also suspect that `Obj.x` in line 21–22 may need to be changed to `Obj.y` but the evidence in these two functions is ambiguous.

Copy-paste bugs yield code pairs that are similar to each other, but the pasted (buggy) code likely has at least one variable usage that is inconsistent with the rest of the code. We search for these bugs in two steps.

Step 1. *Finding Candidate Pairs* — We simply locate pairs of subroutines (P, Q) that are copy-paste candidates from either the input program or the corpus. If one of P and Q contains a copy-paste bug, the two subroutines are likely similar but not identical. We can use n-gram statistics to score subroutine similarity.

Step 2. *Finding Surprising Variable Usage* — We examine each subroutine pair (P, Q) from the above step for a variable usage that is "more surprising" in one subroutine than its counterpart in the other. For example, in Figure 4, lines 9 and 21 are not only textually identical, but they clearly play a similar role in each piece of code. In correct copy-pasted code, the n-grams for lines 9 and 21 should also be similar; for buggy code, the two lines will have dissimilar n-grams. Our system identifies such cases as potential bugs.

5.1 Finding Candidate Pairs

Finding copy-paste candidates is essentially finding heavily plagiarized code snippets. Unlike plagiarism detection (where we want to report any plagiarism no matter how small the plagiarized part is), we want to find copy-paste candidates that are large enough and almost the same. To prune false positives, we focus on finding copy-paste *subroutines*. Therefore we use the following method instead of the plagiarism detector we have developed to find copy-paste candidates. We can use the n-gram data to obtain a function that scores the similarity of subroutine pairs:

$$\alpha(P, Q) = \frac{|\Gamma_\theta^{(n)}(P) \cap \Gamma_\theta^{(n)}(Q)|}{|\Gamma_\theta^{(n)}(P) \cup \Gamma_\theta^{(n)}(Q)|},$$

where P and Q are any subroutines in the corpus. This similarity function is the Jaccard similarity coefficient. It scores all subroutines P and Q in the corpus, locates those where $1 > \alpha(P, Q) \geq t$ (where threshold t is a tunable parameter) and sends them to the next step.

To speed up the process of finding all similar subroutines for an input subroutine P, we built an inverted index that lists, for each n-gram with non-trivial *tf-idf*, the subroutines that contain the gram. When finding candidates for P, we enumerate all of P's n-grams, look each up in the index, and then retrieve the indexed subroutines. We then score only these returned subroutines with the full α function. Finally, we deduplicate results before passing them to the next stage.

5.2 Finding Surprising Variable Usage

Finding surprising variable usage in a subroutine pair is more challenging than simply finding similar subroutines. We identify surprising variable usages in three steps: (1) matching subroutine statements, (2) matching subroutine variables, and (3) scoring the likelihood of copy-paste bugs by counting "surprising" variables.

Matching subroutine statements matches each statement in subroutine P with a "most-similar" statement in subroutine Q. For example, in Figure 4, we want to match lines 9 and 21, because they appear to play the same role, even though the lines of code are not identical. In contrast, no programmer would believe that lines 9 and 23 are at all similar. More formally, let subroutine P consist of canonical statements $\langle p_1, p_2, ..., p_k \rangle$ and let subroutine Q consist of canonical statements $\langle q_1, q_2, ..., q_l \rangle$. A matching consists of statement pairs such that if it contains (p_i, q_j), then p_i plays the same role in P as q_j in Q. We limit ourselves to matchings where matched statements share an operator (*i.e.*, `===` or `while`) and an operand signature (consisting of *variables* vs *constant values*).

Algorithm 2 The heuristic score function to quantify the properness p_i and q_j being matched. It returns a nonzero value if and only if (p_i, q_j) is a valid match.

```
function MatchScore(p_i, q_j)
    if type(p_i) ≠ type(q_j) then
        return 0
    k ← #shared variable names
    s ← 0
    for n ← 1, N do                          ▷ N is the largest level of grams.
        if n-gram(p_i) = n-gram(q_j) then
            s ← 1 + tf-idf(n-gram(p_i)) + k
    return s
```

To find statement matchings that maximize role agreement between each matched pair, we propose Algorithm 2 to score a candidate statement match (p_i, q_j). It is designed to embody two matching heuristics. First, p_i and q_j are more likely to serve the same role if they share more variable names. Second, p_i and q_j are more likely to serve the same role if they share a larger n-gram. For example, consider two different statements q_j and $q_{j'}$ in Q. If the 3-gram of q_j is the same as that of p_i, but only the 2-gram of $q_{j'}$ is the same as that of p_i, then the score of (p_i, q_j) should be higher than that of $(p_i, q_{j'})$. The optimal statement matching $\mathcal{M}_S^*$ is then computed by Algorithm 3 to find the maximum weighted common subsequence with MATCHSCORE as the weight function.

Matching subroutine variables means finding a matching $\mathcal{M}_V$ between pairs of variables—not statements—in subroutines P and Q. (Copy-paste bugs entail variable naming failures.) We exploit the statement matching $\mathcal{M}_S^*$ that was computed in the above step.

We first build a weighted variable mapping graph $G = (V, E)$, in which V consists of the set of all variables in both subroutines. For any variable x in P and y in Q, $(x, y) \in E$ with weight w if (x, y) appears w times in identical operand slots in matched statements $(p, q) \in \mathcal{M}_S^*$. We obtain variable matching $\mathcal{M}_V$ by finding the maximum weighted bipartite matching in G.

Algorithm 3 The statement matching algorithm.

```
function StatementMatching(P, Q)
        ▷ P = ⟨p_1, p_2, ..., p_k⟩, Q = ⟨q_1, q_2, ..., q_l⟩
    for j ← 0, l do
        score[0, j] ← 0                                   ▷ Initialization
        M_{0,j} ← ∅
    for i ← 1, k do
        score[i, 0] ← 0                                   ▷ Initialization
        M_{i,0} ← ∅
        for j ← 1, l do          ▷ Max weighted common subsequence
            s ← MatchScore(p_i, q_j)
            score[i, j] ← score[i − 1, j − 1] + s
            M_{i,j} ← M_{i−1,j−1}
            if s > 0 then                    ▷ (p_i, q_j) is a valid match.
                M_{i,j} ← M_{i,j} ∪ {(p_i, q_j)}
            I ← {(i − 1, j), (i, j − 1), (i, j)}
            (i*, j*) = argmax_{i',j'} {score[i', j'] : (i', j') ∈ I}
            score[i, j] ← score[i*, j*]                   ▷ Pick the best match
            M_{i,j} ← M_{i*,j*}
    return M_{k,l}
```

Identifying copy-paste bugs is the final step. We now have the strongest possible variable matching $\mathcal{M}_V$, as well as the set of all variables ever matched, in the set of edges E. We observe that there might be a copy-paste bug (p_i, q_j) if it contains a variable pair $(x, y) \in E$ that was unusual enough to not be contained in $\mathcal{M}_V$. We thus define the *conflict ratio* [14] of variable x:

$$\gamma(x) = \frac{\sum_{(x,y)\in E-\mathcal{M}_V} w(x, y)}{\sum_{(x,y)\in E} w(x, y)}.$$

If $\gamma(x) = 0$, then all occurrences x in P correspond to $\mathcal{M}_V(x)$ in Q, *i.e.,* there is no copy-paste bug related to x. But if $\gamma(P, Q)$ is very high, it is probable that P and Q were not copy-pasted from one another, or the statements are not correctly matched. Therefore, our system reports a copy-paste bug for x if $\gamma(x)$ is nonzero but small.

6. Experiments

We first describe some details of how we constructed the JavaScript gram corpus and its statistical properties. We then describe our performance in the *plagiarism detection* and *copy-paste bug finding* tasks.

6.1 The JavaScript Gram Corpus

To build the JavaScript program corpus, we scanned the ClueWeb09 web crawl,[2] extracted all JavaScript URLs referenced by the `src` attribute of any `<script>` tags, and retrieved the JavaScript files from the resulting set of URLs. We removed duplicates and finally obtained 2.8M distinct JavaScript files. We used these programs to build our n-gram corpus for $n = 2, 3, 4$ with the method described in Section 3.3.

Most 4-grams can be encoded as strings in seconds. However, some larger 4-grams might take much longer to encode. Therefore, we only generate 4-grams that have 40 or fewer nodes since more than 99% of the 4-grams are smaller than this number. The average size of 4-grams is just 4.62, with variance 8.39. Fewer than 0.005% of the 4-grams failed the encoding. Once all n-grams were encoded, we ran a Hadoop program to compute an *idf* value for each unique n-gram.

In addition to our *programmatic* n-grams, we also built a database of *sequential* n-grams ($2 \le n \le 7$). A sequential n-gram is a sequence of operations of n consecutive statements in a canonicalized JavaScript program; it uses no data or control dependency information. We study sequential n-grams to illustrate how program dependence information can improve the quality of our corpus-driven analysis.

6.2 Plagiarism Detection

In this section we analyze the effectiveness of our corpus-driven filters in improving the accuracy of plagiarism reports from MOSS [25], a widely-used modern plagiarism detector that can process JavaScript programs.

6.2.1 Methodology

A central difficulty in evaluating a plagiarism detection tool is the absence of a ground truth data set, especially for JavaScript. We therefore synthesized test sets of plagiarized programs in the following way:

1. A set S of JavaScript programs was carefully picked from the corpus such that it did not contain any plagiarized code pairs, but included code pairs that looked similar but do not appear to be plagiarized. These similar-but-not-plagiarized code pairs were included in S to fairly evaluate the precision of our plagiarism detectors. The procedure to obtain S is described in Section 6.2.2.
2. Each program in S was then applied a plagiarism technique to produce the plagiarized S'. We used three previously-proposed plagiarism techniques [16], plus one of our own. This step produced plagiarized code pairs for evaluating the recall of the plagiarism detectors. The details of the plagiarism techniques are described in Section 6.2.3.
3. We then submitted the combined set $S \cup S'$ to both standalone MOSS and our filter-based plagiarism detectors. Since we already knew the ground truth of $S \cup S'$, the precision and recall of our plagiarism detectors could be easily evaluated.

6.2.2 Test Sets

The test data was generated in the following way. We first randomly picked a set S of 1,000 JavaScript programs of medium size (containing dozens to hundreds of statements) from the corpus. These programs were then submitted to both MOSS and our plagiarism detectors to obtain hints of possibly plagiarized code pairs. We manually checked the reported suspects, and if a plagiarized code pair was verified, one program from the pair would be removed from

[2] `http://lemurproject.org/clueweb09.php/`

S. All programs triggering false alarms from MOSS and our plagiarism detectors were retained in S, including 10 pairs that looked similar but did not appear to be plagiarized.

We also identified 21 programs from the random set as "commonly written code," which anyone would write independently to achieve certain functionalities. For instance, dozens of consecutive assignments would appear in many object constructors, and should never be considered as plagiarism. Another example for commonly written code is a function to get a cookie value for a specific key. We argue that such code snippet should not be considered as plagiarism since there is a well-known common programming logic to write such a function.

The resulting S contained 212 programs, including 21 programs containing commonly written code. We then manually went through all programs in S to ensure that the set did not contain any plagiarized code pairs.

6.2.3 Plagiarism Techniques

We synthesized the test sets of plagiarized programs using the following three previously-proposed plagiarism techniques [16], plus one of our own.

Identifier Renaming (IR) is simple: just change the names of all identifiers in the original code snippet. **Statement Reordering (SR)** exchanges the order of two statements that do not have data or control dependencies; this method will deceive sequence-based plagiarism detectors but is only possible when the code contains many independent statements. **Code Insertion (CI)** inserts off-topic nonsense statements between real lines of code, thereby "diluting" and breaking up the plagiarized code. Finally, **Code Optimization (CO)** rewrites the original program to be logically equivalent but superficially distinct. This transformation can change the control and data flow, making it very difficult for conventional methods to detect. It is also the only technique we did not derive from Liu, *et al.* [16].

6.2.4 Experimental Setup

We ran standalone MOSS with default parameters as the baseline detector. Our plagiarism detector used MOSS to generate the initial results, then applied our programmatic n-gram post-filters with $n = 2, 3, 4$, and the *tf-idf* threshold $\theta = 6.0$. An n-gram with 6.0 *tf-idf* or higher appears only once in 400 programs on average and thus is quite distinctive. For each of the four test sets, MOSS emitted up to 250 results (and possibly fewer). Because of our test set generation procedure, we know whether each MOSS answer is correct or incorrect. The goal of the plagiarism detection system is to correctly retrieve all the correct answers, and none of the incorrect ones. This method appears in Table 1 as PDG-4GRAM-IDF.

To evaluate the effectiveness of the *tf-idf* measure, we compared against an alternate version of our post-filter that does not use any *tf-idf* information. Put another way, this filter sets its threshold to 0, so that no n-grams are disqualified on the grounds of being too commonplace. This method is thus similar to that of existing PDG-based plagiarism detectors [16]. (We would like to compare our system's performance directly against an extant PDG-based plagiarism detector, but unfortunately we know of no publicly available such detector that can process JavaScript.) This method appears in Table 1 as PDG-4GRAM.

We also examined the utility of program dependence information. The SEQ-4GRAM-IDF method uses tf-idf information that is computed with the conventional *sequential* n-grams. Finally, we tried a mechanism that uses no *tf-idf* information and sequential grams only; it appears as SEQ-7GRAM.

	Precision			Recall		
Set	MOSS	SEQ-7GRAM	Δ	MOSS	SEQ-7GRAM	Δ
IR	73.20%	73.79%	**0.59%**	95.81%	95.81%	**0.00%**
SR	46.00%	49.78%	**3.78%**	98.29%	98.29%	**0.00%**
CI	49.60%	54.63%	**5.03%**	64.92%	64.92%	**0.00%**
CO	64.40%	65.45%	**1.05%**	86.10%	86.10%	**0.00%**
Set	MOSS	PDG-4GRAM	Δ	MOSS	PDG-4GRAM	Δ
IR	73.20%	82.87%	**9.67%**	95.81%	93.72%	**-2.09%**
SR	46.00%	62.50%	**16.50%**	98.29%	98.29%	**0.00%**
CI	49.60%	67.39%	**17.79%**	64.92%	64.92%	**0.00%**
CO	64.40%	73.94%	**9.54%**	86.10%	74.33%	**-11.76%**
Set	MOSS	SEQ-4GRAM-IDF	Δ	MOSS	SEQ-4GRAM-IDF	Δ
IR	73.20%	91.84%	**18.64%**	95.81%	94.24%	**-1.57%**
SR	46.00%	77.24%	**31.24%**	98.29%	95.73%	**-2.56%**
CI	49.60%	75.51%	**25.91%**	64.92%	58.12%	**-6.81%**
CO	64.40%	79.29%	**14.89%**	86.10%	83.96%	**-2.14%**
Set	MOSS	PDG-4GRAM-IDF	Δ	MOSS	PDG-4GRAM-IDF	Δ
IR	73.20%	86.06%	**12.86%**	95.81%	93.72%	**-2.09%**
SR	46.00%	70.44%	**24.44%**	98.29%	95.73%	**-2.56%**
CI	49.60%	73.21%	**23.61%**	64.92%	64.40%	**-0.52%**
CO	64.40%	77.78%	**13.38%**	86.10%	82.35%	**-3.75%**

Table 1: Accuracy of MOSS (baseline) and our tools that apply different filters on MOSS reports. SEQ represents our filter that uses sequential n-gram and PDG represents programmatic n-gram filter. Filters with the suffix -IDF use *tf-idf* values learned from analyzing the corpus to discard unimportant n-grams. The Δ columns show the differences in precision and recall after applying our filters to MOSS output.

6.2.5 Evaluation Results

Table 1 shows precision and recall for standalone MOSS as well as our filter-based systems. In each setting, we choose the value of n in n-gram such that it achieves the best F_1 score,[3] which is commonly used to evaluate systems that emphasize both precision and recall. (Thus, we use 4-grams for all systems except for SEQ-7GRAM.)

As can be seen, all post-filters that use the *tf-idf* information, which we collected from the corpus, are able to obtain large increases in precision across all four test sets. Recall gets worse in all cases, but these decreases are small compared to precision gains, and are especially small in the

[3] $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$

PDG-4GRAM-IDF filter. In other words, F1 scores for all our filters are significantly higher than the F1 score for baseline MOSS reports. Thus, a corpus-driven approach can significantly improve the accuracy of a plagiarism detection tool.

The PDG-4GRAM filter outperformed SEQ-7GRAM, showing that *programmatic* n-grams could yield better accuracy than *sequential* n-grams. However, our PDG-4GRAM-IDF filter did not beat SEQ-4GRAM-IDF in precision. This is because MOSS is itself sequence-oriented. When MOSS incorrectly removed true plagiarized pairs, it made disproportionately more errors for pairs that PDG-4GRAM-IDF would retain (over pairs that SEQ-4GRAM-IDF would retain). If we included all of the false negatives that MOSS removed before applying our filters, on average PDG-4GRAM-IDF would have 75.52% precision and 84.72% recall overall, whereas SEQ-4GRAM-IDF would have 76.12% precision and 83.29% recall overall. Of our four tasks, the **CI** task should present the biggest challenge for sequential techniques, which have no obvious way to avoid the inserted nonsense code. Indeed we can observe that **CI** is the biggest opportunity for programmatic n-grams in our experiments that used both MOSS and the false negatives: the sequential n-grams yield precision of 80.85% and recall of 65.52%, while programmatic n-grams yield a near-identical precision of 80.60% with recall improved to 73.33%.

6.3 Copy-Paste Bug Finder

We evaluate our tool for finding two types of copy-paste bugs. One is *inter-program* copy-paste bugs introduced by copy-pasting code from another program (e.g., a program found in the Internet). The second type is *intra-program* bugs that are introduced by a programmer by copy-pasting from her own code. In total, our system detected as many as 15 previously unknown bugs in source code that we had not previously seen.

6.3.1 Experimental Setup

We randomly chose a set S of 100,000 programs to evaluate our copy-paste bug finder. (Note, we still use the corpus of n-grams that we constructed from 2.8 million programs for this analysis.) We used our tool to find *intra-program* copy-paste bugs in each program in the set S. In addition, we randomly chose a smaller set SS of 5,000 programs from S. We checked for *inter-program* copy-paste bugs between every function in SS and all the functions in the larger set S. We manually examined the reports generated by our tool to distinguish between true and false positives.

We used our corpus-driven code clone detection technique (Section 5.1) to locate copy-pasted subroutines and then used our heuristics to find copy-paste bugs (Section 5.2). We also implemented the heuristics used in CP-Miner [14] to find code clones, and compared the accuracy of our heuristics and theirs.

We used $n = 3$, $\theta = 6.0$, and $t = 0.9$ to locate copy-paste subroutines. We used $N = 4$ (the largest level of grams) when finding variable mappings. We evaluated the effectiveness of our copy-paste bug finder by computing precision and recall for the emitted bug reports. Computing precision is straightforward. For recall, we reported only the absolute number of bugs found, because there is no feasible solution to know all the copy-paste bugs that exist in 100,000 programs.

	Errors reported	Bugs verified	Careless programming	False positives		
				I	II	III
Intra-program	46	11	0	31	0	4
Inter-program	391	18	24	222	49	78

Table 2: The numbers of erroneous subroutines that contain intra-program and inter-program copy-paste bugs.

6.3.2 Evaluation Results

Table 2 shows the numbers of erroneous subroutines that contain *intra-program* and *inter-program* bugs we have found in S. We successfully found 29 previously-unknown bug instances in S. In addition, we also identified some "carelessly written" [14] copy-paste code that has not yet introduced a bug, but could become buggy in the future. For example, one could copy-paste a function and incorrectly rename an identifier to another declared variable. If the expected variable name and the incorrectly renamed variable name both happen to carry the same value, then the program would still execute correctly. However, this fragile code could easily become buggy due to updates in future.

Combining the real bugs and all carelessly programmed code, we achieved 23.9% accuracy for *intra-program* bugs. For inter-program bugs, the accuracy is only 10.7%. One reason is that certain false positives appear several times in our reports when the same code appears in different programs in S. When we remove the redundant false positives, the accuracy goes up to 14.4%. Also, because the same erroneous code could have been copied and used in many different websites, a few true bugs appear more than once in S. Among the 29 true bugs we find 13 of them are unique. The unique bugs and their source is listed in Table 3.

When we applied CP-Miner's heuristics [14] instead of ours, we got 52 reports. It consisted of 1 intra-program bug, 1 inter-program bug, 8 carelessly programmed code snippets, and the rest were false positives. Although a higher accuracy was achieved by CP-Miner's heuristics, much fewer bugs were reported, because CP-Miner only reports an error only if it discovers a suspiciously *unchanged* identifier. This strategy helps CP-Miner to achieve a good accuracy, but sacrifices its recall, and thus limits its usefulness.

Although we found a good number of real bugs, we were expecting much more from a set of 100k programs. After checking the set, we found that about half of the programs are simple JavaScripts generating advertisements, and there are many program-generated programs as well. We believe that we could discover more bugs if we construct a set that contains higher fraction of human-written code.

URL	(Line, Character)	Bug Description
http://www.pets-memories.com/js/wz_tooltip.js	(161, 56)	t_bc should be t_bgc
http://wiki.ext-livegrid.com/chrome/common/js/jquery.js	(23, 13635)	removeAttribute() should be called before overwriting bq
http://talentedtom.liquidpoker.net/calc/codeBehind.js	(1114, 28)	selectedCard should be boardSelected
http://www.vcdd.org.uk/js/slideshowPart1.js	(667, 24)	slide should be thisSlide
http://www.tutvid.com/SpryAssets/xpath.js	(685, 43)	input should be n
http://www.iconutils.com/menu.js	(50, 10)	document should be this.o.ref
http://www.shopconcordmall.com/includes/js/fading_slideshow.js	(138, 29)	obj should be this
http://bostonballet.org/WorkArea/java/webtoolbar.js	(382, 9)	curleft should be curTop
http://www.myradiationsign.com/xp/xp5_searchbar.js	(983, 33)	PARTIALxtablestr should be PARTIALxstr
http://es.tuaviso.net/template/_global/js/drop.js	(593, 32)	tempSelected should be tempSelectedn3
http://minnesota.kudzu.com/content/includes_kudzu/video3/js/util.js	(169, 29)	funcName should be callback
http://www.greenindiastandards.com/includes/pull-down3.js	(646, 1)	top should be left
http://homefreetrade.com/js/floatbox.js	(1715, 7)	this should be document

Table 3: List of unique bugs found in 100,000 randomly selected programs.

6.3.3 Discussion and Failure Types

We were able to classify all but one false positive under three types.

Type I false positives result when the bug finder mismatches two statements in two subroutines. They account for 222 of the 384 false positives. The mismatch may happen because the subroutines are not in fact copy-pasted. It can also happen when the subroutines match, but the statements do not. Figure 5 shows an example of Type I false positive: function `echeck` is a refinement of function `emailCheck`, having added an extra statement at line 8. Our copy-paste bug finder mismatched line 8 to line 3. To fix this, we plan to refine the statement matching algorithm to have stricter conditions on whether two statements can be matched.

Type II false positives result when one subroutine uses the same variable to store several independent values, while the other subroutine uses different variables. They account for 49 of the 384 false positives. These false positives can be eliminated by using a data dependence analysis to remove all the name dependencies by renaming the variables.

Type III false positives happen for various reasons and account for 82 of the 222 false positives. Figure 6 shows one interesting example where the two subroutines contain code snippets that do exactly the same thing in different ways: function `foo` stores the string of a number in variable `h`, while function `bar` stores the number in `h` and the string in `s`. They both use `h < 16` in their `if` statements, which is still correct in `foo` owing to the type coercion rules of JavaScript. We cannot resolve this unless we have a good way to identify that both `h` and `s` hold the same "semantic value" in `bar`.

```
1 function emailCheck (emailStr) {
2   var emailPat = /^(.+)@(.+)$/;
3   var validChars = "[a-zA-Z0-9_-]";
4   ...
5 }

6 function echeck (emailStr) {
7   var emailPat = /^(.+)@(.+)$/;
8   var specialChars
      = "\\(\\)<>@,;:\\\\\\\"\\.\\[\\]";
9   var validChars
      = "\[^\\s" + specialChars + "\]";
10  ...
11 }
```

Figure 5: An example of a Type I false positive bug.

```
1 function foo(rvs) {
2   var h = parseInt(rvs[i]).toString(16);
3   if (h < 16) {
4     h = "0" + h
5   }
6 }

7 function bar(rvs) {
8   var h = parseInt(rvs[i]);
9   var s = h.toString(16);
10  if (h < 16) {
11    s = "0" + s
12  }
```

Figure 6: An example of a Type III false positive bug.

7. Related Work

The most closely-related is natural language processing work that exploits **corpus statistics**. A common way to use corpus statistics in information retrieval is via the standard *tf-idf* formulation [26]. Brants, *et al.* [2] described an n-gram language model that featured a very large Web-derived training corpus and a simple method for smoothing probability estimates in the face of sparse data. Recent research [8, 21, 24] showed evidence that various natural language processing techniques, including the sequential n-gram model, can be applied to programming languages as well for code completion. But to our knowledge, the n-gram model has not been applied to rank the importance of subgraphs in a program dependency graph to aid program analysis. We also contribute by building two program analysis tools based on this information.

Plagiarism Detection — To our knowledge, we are the first to have used large program corpus statistics to weigh the importance of code segments, and apply that for plagiarism detection. Existing plagiarism detection algorithms can be categorized into one of the following types: string-based,

token-based [7, 11, 22, 25], abstract-syntax-tree-based [4, 10], and program-dependence-graph-based [6, 13, 16].

One of the most widely used plagiarism detection systems is *MOSS* [25]. It supports JavaScript. It uses a local fingerprinting algorithm that hashes sets of tokens in a fixed-size window that scans over the entire program text, and improves the results with secret heuristics. It fails to detect plagiarized code that introduces many small local changes (e.g., code insertion). Our program dependence graph based solution addresses this problem. Also, MOSS has no notion of the importance of code segments.

GPLAG [16] transforms program statements into program dependence graphs, and then attempts to find isomorphic subgraphs among these graphs. However, like MOSS, it does not have a notion of the importance of code segments. Green, *et al.* [7] proposed to detect plagiarism by finding programs that share unusual sequential trigrams, where a trigram is more unusual if it occurs in fewer programs. However, like MOSS, this method fails for plagiarized code that is obfuscated with code insertion, because its analysis is not based on a program dependence graph. Also, they derive the importance of a trigram from a small set of programmer specified programs, not a large corpus built from the Web.

Copy-paste Bug Finding — Copy-paste bugs are a known category of software defect [11]. But our system is the first tool that finds copy-paste errors in JavaScript code, and also the first to use corpus-driven program analysis to find potential copy-paste candidates. The most relevant work is *CP-Miner et al.* [14], which finds copy-paste bugs in a large software system. It detects code clones by finding *frequent statement sequences*, and then finds copy-paste bugs through detecting inconsistencies between code clones. It only finds bugs when a variable name remains unchanged by mistake when copy-pasting code. In contrast, our system is capable of detecting erroneous renaming of variable names. Jiang, *et al.* [10] detect code clone bugs by finding inconsistencies between abstract syntax trees using various heuristics. They find code clone bugs that cannot be found by *CP-Miner*, but their overall precision is lower than CP-Miner and our tool.

8. Conclusions

We have proposed a technique for processing large corpora of source code in order to build a statistical summary of different programming patterns. The result is a general-purpose statistical resource that can be applied to programming problems. We demonstrated its utility in finding plagiarized code pairs, and obtained much higher precision rates than a state-of-the-art tool. We also used the statistical resource to locate 29 previously-undiscovered copy-paste programming errors, asking a human observer to examine just a handful of candidates for each true error found.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools: Second Edition*. Addison-Wesley, 2007.

[2] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *EMNLP-CoNLL*, pages 858–867, 2007.

[3] D. Cai and M. Kim. An empirical study of long-lived code clones. In *FASE*, pages 432–446, 2011.

[4] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/24039.24041`.

[6] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 321–330. IEEE, 2008.

[7] P. Green, P. C. Lane, A. Rainer, S. Bennett, and S.-B. Scholz. Same difference: Detecting collusion by finding unusual shared elements. In *Proceedings of the Fifth International Plagiarism Conference*, 2012.

[8] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337322`.

[9] A. Islam and D. Inkpen. Real-word spelling correction using google web 1tn-gram data set. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM, pages 1689–1692, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-512-3. . URL `http://doi.acm.org/10.1145/1645953.1646205`.

[10] 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*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. . URL `http://dx.doi.org/10.1109/ICSE.2007.30`.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. . URL `http://dx.doi.org/10.1109/ICSE.2009.5070547`.

[12] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: a search engine for binary code. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 329–338. IEEE Press, 2013.

[13] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Static Analysis*, pages 40–56. Springer, 2001.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006. ISSN 0098-5589. .

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. . URL http://doi.acm.org/10.1145/1065010.1065014.

[16] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD*, pages 872–881, 2006.

[17] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 378–388, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. . URL http://doi.acm.org/10.1145/1993498.1993543.

[18] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, 2008.

[19] J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, J. P. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak, and E. L. Aiden. Quantitative Analysis of Culture Using Millions of Digitized Books. *Science*, 331:176–, Jan. 2011.

[20] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[21] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. . URL http://doi.acm.org/10.1145/2491411.2491458.

[22] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8:1016–1038, 2001.

[23] M. Ravallion. The two poverty enlightenments: Historical insights from digitized books spanning three centuries. *Poverty and Public Policy*, 3(2):1–46, 2011. ISSN 1944-2858.

[24] 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*, page 44. ACM, 2014.

[25] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD Conference*, pages 76–85, 2003.

[26] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. In P. Willett, editor, *Document retrieval systems*, pages 132–142. Taylor Graham Publishing, London, UK, UK, 1988. ISBN 0-947568-21-2.

[27] L. Thomas, S. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pages 1097–1101, 2006. .

[28] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 976–985, 2007. .

[29] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 335–346, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. . URL http://doi.acm.org/10.1145/1007568.1007607.

[30] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, pages 181–192, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-926-5. . URL http://doi.acm.org/10.1145/1353343.1353369.

[31] L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *Proc. VLDB Endow.*, 2(1):886–897, Aug. 2009. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1687627.1687727.

A. Canonical Transformation

In this section, we describe the function that transforms a JavaScript program to its canonical form. While we handle most common features in the JavaScript language, we do not handle some features such as dynamic scoping and exception handling precisely (Section A.5). As a result, we may miss a few dependencies in the program dependency graph (PDG) we construct. Fortunately, our statistical corpus-driven approach can tolerate these inaccuracies in the PDGs.

The canonical transformation function χ takes a JavaScript statement or an expression as input, and transforms it into a pair $(val, stmt^*)$, where $stmt^*$ is a list of canonical statements that describes the functionality of the input statement or expression, and val holds the result of the statement or expression. Depending on the type of the input statement or expression, val can be an l-value, a constant literal, or `none` is no result is generated.

To understand how the transformation rules are stated, let us start with the following example. Consider the following rule:

$$\frac{\begin{array}{c} expr \rightarrow expr_1\ op\ expr_2 \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_1, S_2, var\ \texttt{=}\ val_1\ op\ val_2\ \texttt{;} \rangle)}$$

The first line specifies the context-free reduction rule that is used to parse the expression. In this case, it says that the above rule is applied when the input expression is a binary operation. The remaining equations above the horizontal line are the preconditions for the rule, and post-condition of the transformation rule is listed below the line. So the above rule states that if $expr_1$ is transformed into (val_1, S_1) and $expr_2$ is transformed into (val_2, S_2), and if we create a temporary variable var through the `NewTemp()` special function, then the resulting canonical statement consists of S_1 and S_2, followed by the statement that assigns the results of $expr_1\, op\, expr_2$ into var.

For convenience, we use the following two special functions to simplify the representation: the `NewTemp()` function that returns an unused name for creating temporary variables, and the `Replace(`$stmt^*$`,`var_1`,`var_2`)` function that replaces var_1 with var_2 in $stmt^*$ and returns the new statements.

A.1 Simple Expressions and Statements

We start with defining the canonical transformation for an expression in JavaScript. If the expression is a variable or a literal (a constant or a function literal), the transformation is straightforward:

$$\frac{expr \rightarrow var}{\chi(expr) = (var, \langle\rangle)}$$

$$\frac{expr \rightarrow literal}{\chi(expr) = (literal, \langle\rangle)}$$

The transformations for the unary, binary and ternary operations are defined as follows. We do not perform any short circuit for logical operators and ternary operators since the goal of this transformation is to preserve semantics, not optimization.

$$\frac{\begin{array}{c} expr \rightarrow op\ expr_1 \\ (val_1, S_1) = \chi(expr_1) \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_1, var\ \texttt{=}\ op\ val_1\ \texttt{;} \rangle)}$$

$$\frac{\begin{array}{c} expr \rightarrow expr_1\ op\ expr_2 \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_1, S_2, var\ \texttt{=}\ val_1\ op\ val_2\ \texttt{;} \rangle)}$$

$$\frac{\begin{array}{c} expr \rightarrow expr_1\ \texttt{?}\ expr_2\ \texttt{:}\ expr_3 \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ (val_3, S_3) = \chi(expr_3) \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_1, S_2, S_3, var\ \texttt{=}\ val_1\ \texttt{?}\ val_2\ \texttt{:}\ val_3\ \texttt{;} \rangle)}$$

The naive way to transform an assignment is to get the result of its right-hand-side expression, and assign it to its left-hand-side expression (which should be an l-value), but this generates redundant dummy assignments since a new temporary variable might be generated by its right-hand-side expression, causing a change to the distribution of the grams. Therefore we check and remove the temporary variable when necessary.

$$\frac{\begin{array}{c} expr \rightarrow expr_1\ \texttt{=}\ expr_2 \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ val_2 \text{ is a variable created by } \texttt{NewTemp()} \end{array}}{\chi(expr) = (val_1, \texttt{Replace(}S_1\texttt{,}val_1\texttt{,}val_2\texttt{)})}$$

$$\frac{\begin{array}{c} expr \rightarrow expr_1\ \texttt{=}\ expr_2 \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ val_2 \text{ is an ordinary variable or a literal} \end{array}}{\chi(expr) = (val_1, \langle val_1\ \texttt{=}\ val_2\ \texttt{;} \rangle)}$$

For compound assignments (such as `+=` and `-=`) and count operations (`++` and `--`), we first transform them into the form of $expr_1$ `=` $expr_1\ op\ expr_2$, then use the above rules to complete the transformation. We omit the rules here.

After the transformations for expressions are defined, we can now show some transformation for simple statements in JavaScript. First, we simply throw all no-ops away (it

won't cause any problem even when the no-op is in an empty loop—see below). And for those statements containing only a single expression, we use the following transformation:

$$\frac{stmt \rightarrow expr\,\texttt{;} \quad (val, S) = \chi(expr)}{\chi(stmt) = (\texttt{none}, S)}$$

A.2 Functions

For the convenience of program dependence analysis, we add a `begin` and an `end` statement in the canonical form of a function:

$$\frac{expr \rightarrow \texttt{function}\ identifier\texttt{(}var^*\texttt{)}\ \texttt{\{}\ stmt^*\ \texttt{\}} \quad (\texttt{none}, S) = \chi(stmt^*)}{\chi(expr) = (\texttt{function}\ identifier\texttt{(}var^*\texttt{)}\ \texttt{\{ begin;}\ S\ \texttt{end; \}}, \langle\rangle)}$$

Anonymous functions without names are similar.

For function call expressions, we can either use a name with an argument list to call a function or write down an anonymous function literal before providing the argument list. Since JavaScript treats a named function as a function object referenced by a variable of the given name, these two syntices of function calls can actually be unified. Hence the transformation of a function call with n arguments is defined as:

$$\frac{\begin{array}{c} expr \rightarrow expr_0\texttt{(}expr_1\texttt{, . . . ,}\ expr_n\texttt{)} \\ (val_0, S_0) = \chi(expr_0) \\ (val_1, S_1) = \chi(expr_1) \\ \vdots \\ (val_n, S_n) = \chi(expr_n) \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_0, S_1, ..., S_n, var\texttt{ = }val_0\texttt{(}val_1\texttt{, . . . ,}\ val_n\texttt{);}\rangle)}$$

Instead of using a stack to manage the arguments in other three address code, we preserve the whole list of arguments in the function call statement. And we enforce every function call to have a variable to store the returned value, even if it has none, to simplify the canonical form.

The `new` expression is similar to function calls and we omit its rule here.

A.3 Field and Array Accesses

In JavaScript, field and array accesses are actually the same thing. The syntices are interchangeable if the index of an array access is known at compile time. However, to retain more high-level code structures provided by programmers, our canonical form distinguishes these two forms of property accesses. Furthermore, the transformation should return a dot or bracket expression as *val* when the field or array access is used as an l-value. We also design it to return the dot or bracket expression if it serves as a function call to preserve the code structure. The rules are presented as follows.

$$\frac{\begin{array}{c} expr \rightarrow expr_1\texttt{.}\,identifier \\ (val_1, S_1) = \chi(expr_1) \\ expr \text{ is an r-value} \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_1, var\texttt{ = }val_1\texttt{.}\,identifier\texttt{;}\rangle)}$$

$$\frac{\begin{array}{c} expr \rightarrow expr_1\texttt{.}\,identifier \\ (val_1, S_1) = \chi(expr_1) \\ expr \text{ is an l-value or a function in a call expression} \end{array}}{\chi(expr) = (val_1\texttt{.}\,identifier, S_1)}$$

$$\frac{\begin{array}{c} expr \rightarrow expr_1\texttt{[}expr_2\texttt{]} \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ expr \text{ is an r-value} \\ var = \texttt{NewTemp()} \end{array}}{\chi(expr) = (var, \langle S_1, S_2, var\texttt{ = }val_1\texttt{[}val_2\texttt{];}\rangle)}$$

$$\frac{\begin{array}{c} expr \rightarrow expr_1\texttt{[}expr_2\texttt{]} \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ expr \text{ is an l-value or a function in a call expression} \end{array}}{\chi(expr) = (val_1\texttt{[}val_2\texttt{]}, \langle S_1, S_2\rangle)}$$

A.4 Control Flow Statements

To simplify the control flow statements, the condition components in these statements must be a single value in their canonical form. Therefore the transformation for the `if` statement is:

$$\frac{\begin{array}{c} stmt \rightarrow \texttt{if (}expr\texttt{)}\ stmt_1\ \texttt{else}\ stmt_2 \\ (val, S) = \chi(expr) \\ (\texttt{none}, S_1) = \chi(stmt_1) \\ (\texttt{none}, S_2) = \chi(stmt_2) \end{array}}{\chi(stmt) = (\texttt{none}, \langle S, \texttt{if (}val\texttt{)}\ \texttt{\{}\ S_1\ \texttt{\}}\ \texttt{else\{}\ S_2\ \texttt{\}}\rangle)}$$

The transformation for the `while` statement is similar, except that we need to update the condition value at the end of the loop body:

$$\frac{\begin{array}{c} stmt \rightarrow \texttt{while (}expr\texttt{)}\ stmt_1 \\ (val, S) = \chi(expr) \\ (\texttt{none}, S_1) = \chi(stmt_1) \end{array}}{\chi(stmt) = (\texttt{none}, \langle S, \texttt{while (}val\texttt{)}\ \texttt{\{}\ S_1\ S\ \texttt{\}}\rangle)}$$

For the `do-while` and `for` loops, since they can be easily transformed to equivalent `while` loops, we apply such transformations before generating the canonical forms:

$$\frac{\begin{array}{c} stmt \rightarrow \texttt{do}\ stmt_1\ \texttt{while (}expr\texttt{);} \\ (\texttt{none}, S_1) = \chi(stmt_1) \\ (val, S) = \chi(expr) \end{array}}{\chi(stmt) = (\texttt{none}, \langle S_1, S, \texttt{while (}val\texttt{)}\ \texttt{\{}\ S_1\ S\ \texttt{\}}\rangle)}$$

$$\frac{\begin{array}{c} stmt \rightarrow \texttt{for (}expr_1\texttt{; }expr_2\texttt{; }expr_3\texttt{) }stmt_4 \\ (val_1, S_1) = \chi(expr_1) \\ (val_2, S_2) = \chi(expr_2) \\ (val_3, S_3) = \chi(expr_3) \\ (\texttt{none}, S_4) = \chi(stmt_4) \end{array}}{\chi(stmt) = (\texttt{none}, \langle S_1, S_2, \texttt{while (}val_2\texttt{) \{ }S_4\, S_3\, S_2\, \texttt{\}}\rangle)}$$

For the `for-in` and `switch` statements, since it changes the code a lot to transform them to their `while` and `if` equivalences and thus loses some high-level structures, we keep these two constructs in the canonical form with their components properly transformed. For statements `continue`, `break`, `return`, we simply keep them the same after their components are canonicalized.

A.5 Limitations

We may miss a few dependencies as we do not handle dynamic scoping (`with` statement). For `try-catch-finally` statements, we only keep the `try` block and the `finally` block, assuming the program would not generate any exception because it is hard to pinpoint where an exception happens based on only the source code. Since most `catch` blocks only do error handling and do not serve an important functionality in a function, we believe this simplification has only very little effect on our analysis.