Semantic Code Search via Equational Reasoning

Varot Premtoon
Massachusetts Institute of Technology
USA
varot@mit.edu

James Koppel
Massachusetts Institute of Technology
USA
jkoppel@mit.edu

Armando Solar-Lezama
Massachusetts Institute of Technology
USA
asolar@csail.mit.edu

Abstract

We present a new approach to semantic code search based on equational reasoning, and the Yogo tool implementing this approach. Our approach works by considering not only the dataflow graph of a function, but also the dataflow graphs of all equivalent functions reachable via a set of rewrite rules. In doing so, it can recognize an operation even if it uses alternate APIs, is in a different but mathematically-equivalent form, is split apart with temporary variables, or is interleaved with other code. Furthermore, it can recognize when code is an instance of some higher-level concept such as iterating through a file. Because of this, from a single query, Yogo can find equivalent code in multiple languages. Our evaluation further shows the utility of Yogo beyond code search: encoding a buggy pattern as a Yogo query, we found a bug in Oracle's Graal compiler which had been missed by a hand-written static analyzer designed for that exact kind of bug. Yogo is built on the Cubix multi-language infrastructure, and currently supports Java and Python.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Theory of computation** → *Abstraction*; *Program specifications*; **Equational logic and rewriting**.

Keywords: equational reasoning, code search

ACM Reference Format:
Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3386001

1 Introduction

In programming, a common and commonly-intimidating task is to discover all places in a codebase which perform some similar operation. For instance, if a common idiom is discovered to have a bug, then all instances of that idiom must be changed. Similarly, if there are many places in the codebase which manually access a data representation, then changing the representation demands not only locating all code which accesses the data structure, but also recognizing which high-level operation they perform. Conventional code search is helpful in identifying these locations, but can still miss a long tail of unexpected variations of the common pattern.

For example, consider an E-commerce app that represents the items in a shopping cart as an unordered array with duplicates. The programmer wishes to find all code that counts the frequency of a given item in the list, as part of some larger refactoring, whether to replace all of them with a shorter or more efficient implementation of frequency counting, or to switch to an alternate representation of shopping carts altogether. Figures 1a and 1b show example code and a refactored version.

Although this is one of the simpler instances of this problem, finding all equivalent code in a codebase is already hard:

1. Code that counts the frequency of items may be *interleaved* with other code, as in Figure 1c. Any tool that can identify this example would hence need to identify it as a *discontiguous match*.
2. The code may be *paraphrased* via syntactic variation or different approaches altogether. Figures 1d and 1e show alternate implementations of frequency-count using different naming, loops, and conditionals.

The variants of array-counting are *semantic clones* of each other. In our experiments, we use our tool to identify more subtle duplication in actual codebases.

Abstracting Away Syntax. How can a program recognize that the examples in Figure 1 are all instances of the same pattern? This has long been the goal of *semantic code search* (for recognizing a concept in a codebase), and the closely-related problem *semantic clone detection* (identifying semantically-similar code). There are a many tools for versions of this problem (§6), each solving its own variation of the problem; they vary in scope of the search (single codebase vs. open web), type of query (natural language, programmatic, test cases), and priorities (fast/imprecise vs. slow/accurate). In this work, we are concerned specifically with code searches intended to help a programmer change a codebase, measured to be 16% of code searches [69]. We

PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3386001

```
count = 0
for a in cart:
  if a == item:
    count += 1
use(count)
```

(a)

```
use(cart.count(item))
```

(b)

```
count = 0
for i in cart:
  if debug:
    print(cart[i])
  if cart[i] == item:
    count += 1
use(count)
```

(c)

```
count = 0
for i in range(len(arr)):
  if itm != arr[i]:
    continue
  count += 1
use(count)
```

(d)

```
count = 0
i = 0
while i < len(cart):
  if cart[i] == k:
    count += 1
  i += 1
use(count)
```

(e)

Figure 1. Variations over the array frequency count pattern in Python

specifically attack an extreme version of the problem, motivated by large refactoring tasks, where every result returned is programmer time saved — or a bug averted. In this extreme, the goal is to exactly match all instances of a semantic concept within a single codebase. To do so, we are willing to spend computation on the most powerful reasoning techniques available. Our results offer a search procedure which is complete in semantic equivalence up to a set of graph-rewrite rules, and which is embarrassingly parallel, able to query a 1.2 million line codebase in 2.5 hours using 30 machines.

In recent decades, the go-to technique for matching code beyond the syntactic level has been *program dependence graphs* [13], used by seven separate tools [15, 22, 24, 34, 36, 39, 85]. PDGs abstract away the exact ordering of statements in favor of identifying when one statement uses the result of another (data dependence) or is guarded by a condition (control dependence). In doing so, they can detect when statements of interest are interleaved with other code. However, except for hardcoded special-cases, they are unable to deal with other isomorphisms such as arithmetic identities or alternate kinds of loops, let alone alternate APIs. Other approaches, discussed in §6, similarly fall short.

As a result, we are aware of no prior automated tools that, given the directive to search for array frequency counts, could match all 5 examples in Figure 1 without also incurring a large number of false positives. At one end of the spectrum, a PDG-based tool told to search for the code in Figure 1a may also detect the interleaved code of Figure 1c, but would be stymied by the differences of Figure 1e. At the other end of the spectrum, a `grep`-user could at best try a set of string queries like `count` or `for .* in cart`, and would still have to sift through a large number of false-positives. We revisit this claim in §6, backed by discussion of 70 prior tools.

Yogo: You Only Grep Once. In this paper, we present a new approach to semantic code search based on dataflow equivalences, and our Yogo tool built on it. Yogo takes as input the code to search, a library of pre-written rewrite rules, and a high-level concept expressed as a dataflow pattern. Using a fusion of techniques from Tate et al's *equality saturation* [80] and the Programmer's Apprentice [64], it is able to recognize when a code fragment is equivalent to one of many implementations of said concept.

From equality saturation, we borrow the Program Expression Graph (PEG) representation. Like program dependence graphs, Program Expression Graphs ignore statement ordering and can match patterns interleaved with other code. However, they go further by representing all of a program's semantics, including mutation and loops, as pure data-flow. In doing so, it becomes possible to discover equivalent fragments by applying low-level equations and rewrite rules, and then compactly represent all such equivalent fragments as a structure called an *e-graph* [10, 50, 51]. The result is an efficient procedure for discovering if a program contains a subprogram equivalent to the search pattern. And, if the rules given to the system are sound, and they only entail a finite number of equivalent programs, then the search procedure is sound, and complete with respect to the rules. In our experiments, given the default rewrite rules, Yogo's search terminates in under 3 minutes for over 99% of methods.

From the Programmer's Apprentice, we borrow the idea that high-level concepts can be identified as dataflow patterns. Based on this idea, we can e.g.: create rewrite rules that recognize many implementations of the concept of "iterating through a sequence." A dataflow pattern for array frequency-counting can then use this concept as a subnode, so that it may match any of the varieties of iteration in Figures 1 and 2. The same idea allows our approach to recognize when a program accomplishes the same goal through an alternate API, or even a different algorithm. With this switch to high-level concepts, our technique can even abstract away language-dependent features. The upshot is that, from a single query for array-count frequency, Yogo can recognize not only all five Python variations in Figure 1, but also the three Java variations in Figure 2.

Overall, we make the following contributions:

- A new technique for semantic code search based on dataflow equivalences, which enables a single search query to soundly match highly-dissimilar yet equivalent code fragments, even across languages.
- The Yogo tool implementing this technique.

```
int count = 0;
for (Item x : list)
  if (x == k)
    count += 1;
use(count);
```

(a)

```
int count = 0;
for (i = 0; i < list.size(); i++)
  if (list.get(i) == k)
    count += 1;
use(count);
```

(b)

```
int count = Collections.frequency(list, k);
use(count);
```

(c)

Figure 2. Java variations of array frequency count

```
if (x >= end || x < pos) {
  doSomething();
}
```

(a)

```
if (x >= rect.getLeft()) {
  if (x < rect.getRight()) {
    doSomething();
  }
}
```

(b)

```
boolean b = i >= left;
int j = i;
boolean c = j < right;
if (!(b && c)) {
  doSomething();
}
```

(c)

```
if 10 <= x < 20:
  doSomething()
```

(d)

Figure 3. Example 1D bounds checks

2 Overview

In this section, we demonstrate how to use Yogo to search for a 1-dimensional bounds-check equivalent to `i < lo || i >= hi`. Although small, this is already challenging. Figure 3 shows 4 examples of bounds-checking code, including examples in both positive and negated form, in both Java and Python, and including variants which have no tokens in common with the given pattern.

To search for a bounds check, the user writes a query in Figure 5, using Yogo's DSL[1]. This query is a textual form of the dataflow graph pattern in Figure 6a.

The user then invokes Yogo, indicating the target language and files, the query, and **a standard library of rules**.

```
./yogo java "general_rules.yogo,java_rules.yogo"
        query.yogo *.java
```

The general rules library includes rules for reasoning about boolean and comparison operators, such as the one in Figure 7, which implements De Morgan's law $a \vee b = \neg(\neg a \wedge \neg b)$.

[1]Yogo's DSL is more verbose than the concrete syntax to simplify parsing in the implementation. This is not fundamental to the approach.

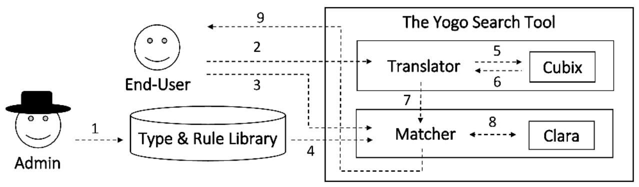

Figure 4. The organization of the Yogo Search Tool and its deployment. The system admin maintains a long-term library of rules and custom types (1), which are reused in every search session (4). Then for each search session, the end-user provides source files (2) and search patterns (3). The tool outputs match results to the end-user (9).

The Java- and Python-specific rule libraries contain rules for reasoning about language-specific constructs and APIs, and for mapping them into language-generic concepts. For example, there is the Python rule giving the isomorphism between `(a <= b < c)` and `(a <= b and b < c)`[2]. Over time, a small set of power-users can add rules for reasoning about new libraries and domains, enabling a large set of end-users to rapidly construct deep semantic queries. Figure 4 gives an overview of how the two kinds of users interact with Yogo.

During execution, Yogo constructs a Program Expression Graph for each method under search. (Yogo only accepts intraprocedural queries.) It then runs its **equality saturation** engine to turn that PEG into an *equivalence graph* on this PEG, or E-PEG, which represents both the original method as well as all methods which can be shown equivalent using the provided rules library.

For example, we show how Yogo matches the query against the code `x >= lo && x < hi`. Yogo first translates this code into the PEG in Box 1 of Figure 6b. After matching the rule for De Morgan's law, Yogo's equality saturation engine extends the PEG with the new nodes in Box 2, and adds dashed equivalence edge between the `and` and `not`. Finally, it matches two rules run witnessing both directions of the equivalences $\neg(a \geq b) = (b < a)$, adding the nodes of Box 3. The entire e-graph in Figure 6b now represents 5 variations of the original code. The or node is equivalent to the search query, and hence Yogo returns a match, with the query variable `root` bound to the or node.

Other rules in Yogo's standard libraries give it the ability to reason about nested conditionals, memory, and assignments. Using these rules, it can expand the four programs of Figure 3 into E-PEGs that compactly represent the exponentially large spaces of equivalent programs, and discover that all four of them contain a 1-dimensional bounds-check.

[2]The purely-functional PEG representation renders short-circuiting and duplication a non-issue, at the cost that PEGs cannot always be mapped back into code.

```
(defsearch bound-checking
 (root <- (generic/binop :or (generic/binop :< x lo)
                             (generic/binop :>= x hi))))
```

Figure 5. A search query for bounds-checking

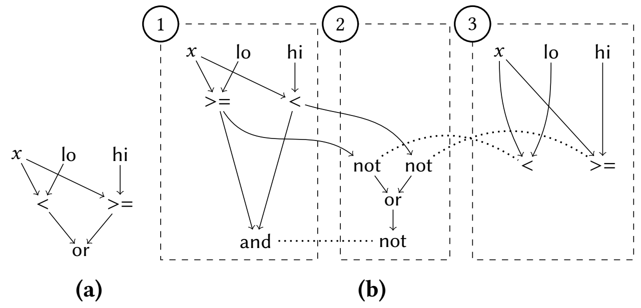

Figure 6. (a): PEG for the query in Figure 5. (b): E-graph representing 5 programs equivalent to `x >= lo && x < hi`, with nodes grouped by order of discovery. Dashed lines are between equivalent nodes. Some nodes duplicated for clarity.

```
(defeqrule demorgan1
 (generic/binop :and a b)
 =>
 (generic/unop :not (generic/binop :or (generic/unop :not a)
                                       (generic/unop :not b))))
```

Figure 7. Yogo rule for one direction of De Morgan's law

Of course, our worked example only shows Yogo reasoning about pure code, using the classic techniques of e-graphs and congruence-closure [10, 50, 51]. In order to scale to all the examples of Figure 3, Yogo must translate stateful code to a form amenable to equational reasoning. §3 explains how Yogo leverages E-PEGs [80] to handle stateful and loopy code, and the insights of the Programmer's Apprentice [64] to match high-level concepts rather than specific code.

3 From Code to Concepts

§2 demonstrated how our approach can discover a pattern in equivalent pure code. In this section, we show how Yogo extends equational reasoning to mutable state and loops, and even to high-level concepts.

In the remainder of this section, we will show examples both using Yogo's DSL, and as general mathematical rules. For convenience, we will refer to steps of the algorithm as actions of Yogo. However, other than specific choices of what nodes may be in the graphs, everything in this section is part of our general approach.

Table 1. Basic nodes and example encodings

Node	Denotation
$Q(\sigma, i)$	$\sigma(i)$
$\text{assign}(\sigma, i, e)$	$(\sigma[i \mapsto e], e)$
$\text{mem}((\sigma, e))$	σ
$\text{val}((\sigma, e))$	e
$\text{sel}(a, i)$	Address of $a[i]$
$\text{fcall}(\sigma, f, \overline{e})$	$(\sigma', v)^*$

* Where (σ', v) are the result of evaluating f on arguments $\overline{e}$, starting in memory state σ.

(a)

Code: `x = y; z = x;`
PEG:
$\lambda\sigma.\textbf{let } r =$
$\quad \text{assign}(\sigma, \text{“}x\text{”}, Q(\sigma, \text{“}y\text{”}))$ **in**
$\text{assign}(\text{mem}(r), \text{“}z\text{”},$
$\qquad Q(\text{mem}(r), \text{“}x\text{”}))$

Code: `arr[i+1]`
PEG:
$\lambda\sigma.Q(\sigma,$
$\quad \text{sel}(Q(\sigma, \text{“arr”}),$
$\qquad Q(\sigma, \text{“}i\text{”}) + 1))$

(b)

Table 2. Loop-related nodes

Node	Denotation
$\text{loop}(e_0, e)$	$\lambda i.\begin{cases} e_0^* & i = 0 \\ e(i-1) & \text{otherwise} \end{cases}$
$\text{final}(e, l)$	$\lambda i.l(\min_{j \in \mathbb{N}} e(j) = \textbf{false})$
$\text{seq}(i, k)$	$(i, i + K, i + 2k, \ldots)$
$\text{iterV}((t_0, t_1, \ldots))$	$\lambda i.\begin{cases} t_i & t_i \neq \bot \\ \bot & \text{otherwise} \end{cases}$
$\text{iterP}((t_0, t_1, \ldots))$	$\lambda i.\begin{cases} \textbf{true} & t_i \neq \bot \\ \textbf{false} & \text{otherwise} \end{cases}$

* Actually $e_0(0)$, but $e_0(i)$ should be invariant in i.

3.1 Memory and Assignments

Yogo's treatment of mutable state is standard: stateful operations such as assignment consume and produce a memory state, denoted σ. Table 1a gives the PEG nodes for state and a simplified version of their denotations, while Table 1b gives example translations of mutable programs into PEG. The most important node is the Q ("query") node for memory lookup. Note that the denotations are only given to define the correctness of equational rules; Yogo never needs to evaluate a PEG.

In our explanation so far, nodes each represent a single value, and hence a fixed-size graph cannot represent a program with loops. In the next section, we introduce the big insight allowing PEGs to represent loopy programs. The result is a referentially transparent yet complete representation of programs, enabling equational reasoning on loops.

3.2 Loops, Conceptually

In this section, we introduce how Yogo is able to reason about code with loops and mutation, and recognize a higher-level concept. We demonstrate how Yogo can model a loop of the form `i = 0; while ...: i += 1`, and recognize the higher-level concept of iterating over the infinite sequence $0, 1, \ldots$.

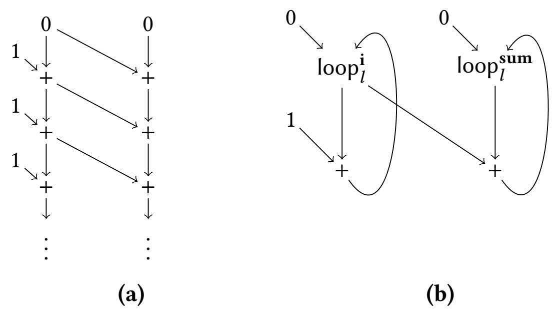

Figure 8. (a) Unfolded dataflow graph of loop body of `sum = 0; for(i = 0; ...; i++){ sum += i; }`. **(b)** PEG for said loop body.

Other queries may be written in terms of this higher-level operation, so that they may match any kind of iteration.

Background: Loops in PEGs. We first explain how PEGs model loops, and then show how Yogo leverages this to recognize higher-level kinds of loops. Consider the loop: `sum = 0; for(i = 0; ...; i++)sum += i;` . Without a static bound on the maximum number of iterations, a dataflow graph for this loop would need an infinite number of nodes, including one each for `i` and `j` in each iteration of the loop. Such a dataflow graph is shown in Figure 8a. Yet each unrolled layer follows a regular pattern, and so it is possible to fold this infinite graph into the PEG in Figure 8b.

We now introduce the final ingredient of PEGs: every node of a PEG represents not a single value, as in a vanilla dataflow graph, but rather a potentially-infinite sequence of values. More formally, every node represents a value of type

$$\mathsf{LoopIndex} \longrightarrow \mathsf{Val} + \bot$$

where LoopIndex is vector of natural numbers, indicating the iteration index of each loop. In this explanation, we assume there is only one loop, so that LoopIndex devolves into a single natural number. The more general version is elaborated in Tate et al [80].

So, in Figure 8b, $\mathsf{loop}_l^{\mathbf{i}}$ and $\mathsf{loop}_l^{\mathbf{sum}}$ represent the values of `i` and `sum` in each iteration of the loop. The l subscript is a loop label which indicates they are sequences of values in the same loop.

Table 2 gives the denotations of Yogo's loop-related nodes. The two primordial ones are loop and final. $\mathsf{loop}(e_0, e)$ represents the infinite sequence whose first element is e_0, and whose successive elements are computed by e. Intuitively, $\mathsf{final}(e, l)$ gives the end result of loop l, by finding the first iteration where e is false, and giving the corresponding value of the sequence produced by l. For example, consider the PEG $\mathsf{final}(\lambda i.i < 2, e)$ where e is defined circularly as $e \leftarrow \mathsf{loop}(3, +(e, 5))$, which represents the final value of n in the program `for(i = 0, n = 3; i < 2; i++, n += 5);`.

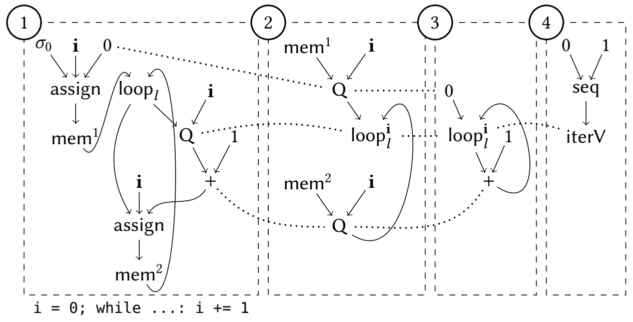

Figure 9. E-graph for `i = 0; while ...: i += 1`. Some nodes duplicated for clarity.

$$\begin{aligned}
\mathsf{final}(\lambda i.i < 2, e) &= e(\min_{j\in\mathbb{N}}(\lambda i.i < 2)(j) = \mathbf{false}))\\
&= e(2) = \mathsf{loop}(3, +(e,5))(2)\\
&= +(e,5)(1) = e(1) + 5(1)\\
&= \mathsf{loop}(3, +(e,5))(1) + 5\\
&= +(e,5)(0) + 5\\
&= \mathsf{loop}(3, +(e,5))(0) + 5 + 5\\
&= 3 + 5 + 5 = 13
\end{aligned}$$

Typically, the initial translation of a program will yield a loop node representing an infinite sequence of memory states. From this, Yogo's equational reasoning engine can discover loop nodes that represent the sequence of values taken by a single expression within the loop, partially independent from the rest of the loop.

The PEG treatment of loops is counterintuitive, but powerful in its ability to bring equational reasoning to loops. For example, this rule pushes memory lookups inside a loop:

$$\mathsf{Q}(\mathsf{loop}(init, next), \lambda) \Longrightarrow \mathsf{loop}(\mathsf{Q}(init, \lambda), \mathsf{Q}(next, \lambda))$$

Discovering a counter. We now turn to how Yogo recognizes that the loop `i = 0; while ...: i += 1` is an instance of the higher-level concept of a sequential counter, giving our first instance of nodes representing abstract concepts, and showing how our use of PEGs surpasses its previous applications in compiler optimization. Box 1 of Figure 9 shows the initial PEG for this code.

Yogo's equality saturation engine then begins applying rewrite rules to obtain the e-graph of equivalent programs. From the initial PEG in Box 1, the following two rules match, yielding the nodes in Box 2. The first rule states that, if a program assigns the value x to l-value λ and then attempts to read λ from the resulting memory, the result is x. The second rule states that memory lookups distribute over looping. It is used to push the Q node inside of the loop, yielding the $\mathsf{loop}_l^{\mathbf{i}}$ node of Box 2, which represents the sequence of values of $\mathbf{i}$ within the loop.

$$\mathrm{Q}(\mathrm{mem}(\mathrm{assign}(\sigma, \lambda, x)), \lambda) \Longrightarrow x$$
$$\mathrm{Q}(\mathrm{loop}(init, next), \lambda) \Longrightarrow \mathrm{loop}(\mathrm{Q}(init, \lambda), \mathrm{Q}(next, \lambda))$$

Thus far, all nodes described have corresponded directly to something in the programming language. We now introduce the first nodes representing higher-level concepts.

The node $\mathrm{seq}(a, k)$ represents the infinite sequence of values $a, a + k, a + 2k, \ldots$. The node $\mathrm{iterV}(\mathrm{seq}(a, k))$ then represents a loop-varying value equal to a in the first iteration, $a + k$ in the second iteration, etc. With these, we can now state a rule identifying a loop which increments a value as an instance of iterating through a sequence. We have redrawn the $\mathrm{loop}_l^{\mathbf{i}}$ node in Box 3 to make it more apparent how the rule matches this node, yielding the final node in Box 4.

$$\mathrm{loop}_l(a, +(i, k)) \Rightarrow \mathrm{iterV}_l(\mathrm{seq}(a, k))$$

There are also similar rules which recognize other instances of looping over a sequence, such as a Python's `for i in range(d):` .... Other patterns may now be written in terms of iterV, and will be able to match any loop over a sequence, including across languages.

3.3 Auxiliary Facts

Many search patterns have *side conditions* which cannot be expressed as another pattern match. For example, loosely speaking, array frequency counts contain an expression equivalent to `arr[i] == k`, but with the extra condition that `k` may not change during the loop.

Our solution is to include in the PEG annotation nodes representing *auxiliary facts* which do not take part in the computation, but may nonetheless be inferred by rewrite rules and used as conditions for other rules. For the problem above, for example, the annotation $\textsc{invariant}(l, e)$ means that e is an expression which does not change between executions of loop l. Two other important examples are the $\textsc{pure}(f)$ annotation, which indicates the auxiliary fact that memory state output by a call to function f is the same as the one input, and the $\textsc{independent}(\lambda_1, \lambda_2)$ annotation, which indicates that a write to l-value λ_1 cannot affect λ_2, and vice versa. (An example of dependent l-values: one l-value representing an entire array, and another representing a single entry in that array.)

Users can program rules to infer auxiliary facts similar to any other equational rule, allowing the user to augment queries with information computed by any terminating term-rewriting system. Auxiliary facts also provide an interface for Yogo rules to ingest information from arbitrary external analyzers.

3.4 All the Array Counts

We now have all the ingredients to show how Yogo can use a single query to match all 8 Java and Python variants in Figures 1 and 2. We first present a query that matches the 6 variants which do not use library functions, and then explain how to add domain knowledge about these functions.

The query for these 6 variants is given below in mathematical notation. Figure 10 shows how it is written in Yogo's DSL.

$$\begin{aligned}
\mathrm{counter} &\leftarrow \mathrm{loop}_l(0, \mathrm{next}) \\
\mathrm{next} &\leftarrow \mathrm{cond}(\mathrm{iterV}_l(\mathrm{coll}) = k, \\
&\qquad \mathrm{counter} + 1, \\
&\qquad \mathrm{counter}) \\
\mathrm{answer} &\leftarrow \mathrm{final}_l(\mathrm{iterP}_l(\mathrm{coll}), \mathrm{counter}) \\
&\qquad \textsc{invariant}_l(k)
\end{aligned}$$

We explain this query in parts. The first line says that the matched PEG must have a node, which we refer to as counter, which represents a sequence that starts at 0, and whose successive values are given by the formula for next.

The second part says that the node labeled next must be a cond node. The guard of this condition must be a comparison between some node k and the current element of the sequence coll. If the condition is true, then the resulting value of the cond node is 1 plus the current value of counter, and is otherwise the current value of counter. The iterV node may be identified from one of many varieties of loop by a process like the one described in the previous section.

The third part looks for the value of the counter at the end of the loop. $\mathrm{iterP}_l(\mathrm{coll})$ is an analogue of iterV which returns true if the element of coll for the current iteration of loop l exists, and false otherwise. As counter represents a sequence of values, $\mathrm{final}_l(\mathrm{iterP}_l(\mathrm{coll}), \mathrm{counter})$ is the value of counter when the loop ends.

Finally, the annotation $\mathrm{invariant}_l(k)$ represents the auxiliary fact that the expression k, which represents the item being counted, must be one which does not change between iterations of the loop l.

Figure 11 gives the initial PEG for the frequency-count snippet in Figure 1e. Even before applying any rewrite rules, several portions already resemble the search pattern.

We now explain how to define a query that also matches the language-specific function calls. We define a new concept node, like the existing concept nodes iterVand iterP, for array counts. We then add three rewrite rules: one that rewrites a match of the previous query to this concept node, and two more that do likewise for Python's `array.count()` and Java's `Collections.frequency()` functions.

Although this requires hardcoding knowledge about the Python `count` and Java `frequency` functions, it is still useful for two reasons. First, this knowledge needs only be recorded

```
(defsearch array-element-count-v1
  (answer <- (generic/final l bound counter))
  (counter <- (generic/loop l (generic/const 0) next))
  (next <- (generic/cond p inc counter))
  (inc <- (generic/binop :+ counter (generic/const 1)))
  (p <- (generic/binop :== e k))
  (e <- (concept/iter-v l coll))
  (bound <- (concept/iter-p l coll))
  (invariant l k))
```

Figure 10. Yogo DSL query for an iterative array count

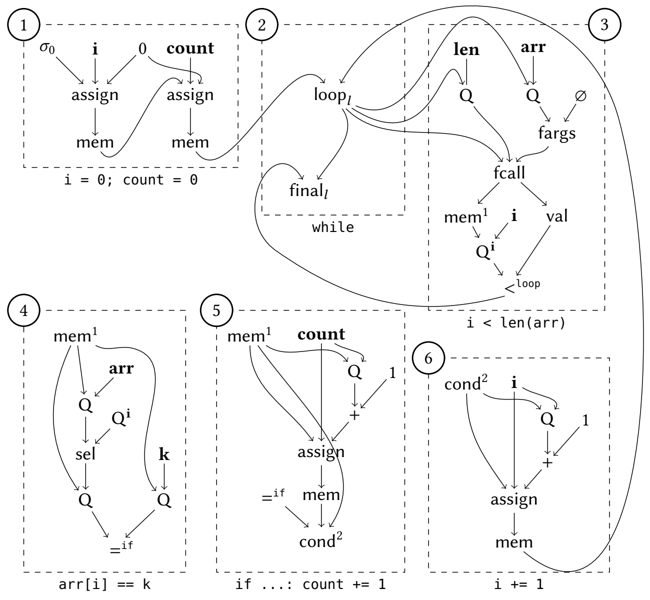

Figure 11. A PEG for the array frequency count code in Figure 1e. Some nodes are duplicated in the illustration for readability, with superscripts to identify them. Only memory states are shown as loop or conditional nodes.

once, and can then be shipped as part of a library of rules. Second, the array-count concept can, in turn, be used in a higher-level concept or search. The query in Figure 12, for instance, searches for code which implements the rule that there may be at most four cards of one type in a deck, seen in collectible card games such as Magic: The Gathering. It can match code which uses any implementation of array frequency counting, which in turn may use any implementation of iterating through a sequence.

```
(defsearch check-at-most-four
  (generic/binop :> (concept/array-count arr k)
                    (generic/const 4)))
```

Figure 12. A query higher in the concept hierarchy

3.5 Additional Discussion

Soundness. Defining the soundness of low-level rewrite rules is simple: the RHS must be semantically equivalent to the LHS. Defining the soundness of a rule that recognizes a concept is less-so. If two different languages have their own functions for file I/O, is it really safe to identify them both with common concepts of file operations?

There are many ways to formalize an answer to this conundrum, but a simple one is this: To say that code implements a concept is to say that the set of behaviors of the code is a simulation refinement [43] of the behaviors of the concept's specification. The function from states of the implementation to states of the concept's specification then induces an equivalence relation that ignores the minute differences in the implementations of that concept.

So, a rewrite rule need only be sound with respect to the equivalence relation of its corresponding concept. Then the library of rewrite rules is collectively sound under the union of these equivalence relations, and thus the search procedure is sound in the transitive closure of the union of these equivalence relations.

Comparison with Tate et al. Yogo features several extensions and design differences to the original presentation of PEGs in Tate et al's system, Peggy [80]. The big conceptual difference is Yogo's coarser notion of equality to support hierarchical abstraction a la the Programmer's Apprentice. This motivates many differences in DSL design. For instance, Yogo has the seq node, whose denotation is an infinite sequence, and is used to define the abstract notion of iterating through a sequence. All of Peggy's nodes, in contrast, have lower-level denotations. While Peggy is limited to a fixed set of equational rules, Yogo offers user-definable one-way rewrite rules, which are used to implement both auxiliary facts and higher-level concepts. Another notable difference in design is Yogo's richer language of l-values.

Rewriting: The Elephant in the Room. Given the utility of Yogo for refactoring and program repair purposes, an obvious question is: is it possible to automatically rewrite the code found by Yogo searches? Unfortunately, the very feature that makes Yogo so enticing, its flexibility in matching, also makes rewriting matches an extremely difficult problem. There are three challenges:

1. Matches may have little syntactic resemblance to the search query. In the example of Figure 6, the matched

code has no non-leaf nodes in common with the bounds-check query.
2. Matches may be interleaved with non-matching code. Even the basic bounds-checking query can match `b1 = (x < lo); b2 = (x > hi); do_something(); b1 or b2`.
3. Matches may be overlapping. For example, when searching for the squared magnitude $(x_1 - x_2)^2 + (y_1 - y_2)^2$, every occurrence will actually result in two matches, because the commuted version $(y_1 - y_2)^2 + (x_1 - x_2)^2$ is also a match.

We thus leave solving these problems to future work. The few existing tools that encounter these problems fare no better. For instance, in §6, we discuss the rewriting tool Coccinelle. Although it has limited ability to match across isomorphisms, it does not address the problems mentioned here, and hence can produce poor output.

Other Limitations. Here are three other limitations of Yogo:

- The Yogo DSL given does not have a good way to express arbitrarily-nested expressions. For example, it is not straightforward to write a constraint "`x` is any argument in a call to `f`." The query in §5.2 does exactly this, but it does it by manually constructing a term-rewriting system to infer this, which is not modular, and loses some of the benefits of Yogo.
- The current Yogo implementation has a very poor solution to the frame problem: if there is a call to a function not known to be pure, Yogo assumes no relation between the memory state before and after the call. There are many simple improvements that can aid this. For example, the current Yogo implementation is not even aware that calls to a different function cannot modify local variables in Java and Python.
- Yogo cannot handle interprocedural patterns. This limitation is more fundamental. While it may be easy to extend Yogo to have all its searches include depth-1 inlining, and Tate et al's Peggy has some support for inlining, allowing arbitrary inlining can explode the size of the E-PEGs.

4 Implementation

Yogo is implemented in 800 lines of Clojure and 2000 lines of Haskell. The Haskell portion defines translators from Python and Java to PEGs, and is based on the Cubix multi-language infrastructure [35]. The Clojure portion defines a DSL for rewrite rules and queries, as well as an equality saturation engine based on the Clara implementation of the Rete algorithm [14]. Yogo comes with 300 lines of generic rewrite rules, 150 lines of Java-specific rules, and 200 lines of Python-specific rules. Yogo uses a heuristic analysis based on method names for inferring method purity, e.g.: it assumes that Java methods starting with "get" or "to" are pure.

5 Evaluation

In this Section, we set out to prove Yogo's ability to search real codebases and to find paraphrases and discontiguous matches in multiple languages. §5.1 presents our systematic study of 9 search patterns, gleaned from a mixture of author suggestions and systematic collection from StackOverflow, on a corpus of 3 codebases. As this evaluation involves artificial searches on small (under $60K$ LOC) codebases, we follow this with our case study on Oracle's Graal project (§5.2), where a Yogo query discovered a bug in a 1.2M LOC codebase. We then discuss why we did not compare Yogo directly to existing tools (§5.3).

5.1 General Patterns, Multiple Codebases

In this section, we develop 9 search patterns independent of language and codebase, and evaluate Yogo's ability to find them in 3 real codebases. The patterns are named in Table 4, and full descriptions and code for the patterns are given in the accompanying technical report [58].

The 9 patterns came from two different sources. The first five, SP1–SP5, were motivating examples in the development of Yogo, and include both the array-frequency and bounds-check examples given in §2 and §3. We obtained our next four from StackOverflow using the following methodology, inspired by a previous study [18].

1. Take the 75 highest voted questions tagged "java," and likewise for "python."
2. Of these, only consider "How to" question. This, excludes, e.g.: "Difference between wait() and sleep()."
3. Among the how-to-do-X questions, only consider ones where the operation X:
   - can be described semantically as values being computed, not what the code looks like or how the values are computed.
   - can be generalized to both Java and Python. This filters out language-specific questions about libraries, Android, conversions between types that do not make sense in both languages, etc.
   - is more than a single operation over the inputs in both languages. This is to avoid having a lot of search patterns that are too simple and have no structure to highlight Yogo's multi-language capability. For instance, we ignore "how to concatenate two lists in Python" and "What's the simplest way to print a Java array" (which is `print arr` in Python).
   - does not directly involve language constructs which the current Yogo translator overapproximates, such as breaks, list comprehensions, and lambdas.
4. Narrow down the remaining questions to those whose accepted answer contains an intraprocedural snippet that does X.

Table 3. Codebases searched, and number of time-outs

			Methods		Non-methods	
	Lang.	LOC	Total	TO	Total	TO
PyGame	Python	49k	2345	21	481	3
Cocos	Python	53k	2876	2	1082	0
LitiEngine	Java	59k	3625	28	997	2

Table 4. The number of matches found for each search pattern and codebase.

Pattern	Cocos2D	PyGame	LitiEngine
SP1. Bound checking	4	3	13
SP2. Squared 2D distance	8	8	10
SP3. Put-if-not-present	3	4	11
SP4. Frequency count	0	0	0
SP5. Time elapsed	1	20	1
SP6. Loop index	120	126	42
SP7. Dictionary iteration	17	6	9
SP8. MD5 hashing	1	1	0
SP9. File writing	8	7	0

5. Rewrite X in a language-generic term (e.g., "Iterate through a HashMap" becomes "Iterating over a [language-generic] dictionary").

Codebases. We use YOGO to search over 3 real-world open-source projects: PyGame, Cocos2D, and LitiEngine. All three are open-source frameworks for creating games. PyGame is written in Python with 49K LOC (commit `ad681aee`). Cocos2D is also written in Python with 53K LOC (commit `9bb2808`). LitiEngine is written in Java with 59K LOC (commit `c188504`). We chose game engines because we expected them to have plenty of meaningful intra-procedural computations, and because two of the search patterns are geometric computations. Table 3 gives details on these codebases. Using the Docker container for YOGO shipped in the accompanying artifact, on one author's computer, YOGO searched PyGame, Cocos2D, and LitiEngine in 3.5, 1, and 4.5 hours respectively.

Timeout. A long source function generally results in a large graph, which may take the rule engine too long to saturate it with equalities. For both Java and Python, we set a time limit of three minutes per graph to perform equality saturation. If a method times out, that means that YOGO has not expanded the E-PEG to contain all programs equivalent under the rules, but YOGO may nonetheless find some matches in the partially-saturated E-PEG. Table 3 shows that YOGO times out on fewer than 1% of methods.

Results. Table 4 reports the number of matches that the matcher finds for each pattern and codebase.

5.1.1 Inspection of Results.

In the following paragraphs, we inspect our results for correctness, search for false positives and false negatives, and share examples of matches to illustrate the power of YOGO.

Examination of Matches. We manually examined all found matches, except for SP6, where we sampled 10 from each codebase. For SP3, SP5, SP6, SP7, SP8, and SP9, all matches are correct. No matches were found for SP4 in any codebase.

For SP2, it is worth noting that each code snippet results in at least two matches because of commutativity. We found that several matches in PyGame actually implement a 3D magnitude rather than a 2D point-distance. Specifically, they compute the magnitude of a vector cross product, and look like this:

```
cross = ((self.v1.y * v.z - self.v1.z * v.y) ** 2 +
        (self.v1.z * v.x - self.v1.x * v.z) ** 2) +
        (self.v1.x * v.y - self.v1.y * v.x) ** 2)
```

While it can be argued that this subcomputation is indeed a 2D magnitude and hence a point distance, it is not intended as a point distance between `(v1.y*v.z, v1.z*v.x)` and the point `(v1.z*v.y, v1.x*v.z)`. This highlights a downside of our approach, which is that it can be challenging to write a search pattern that precisely captures the intent. In this case, the pattern searches for any computation of the form `(x1 - x2)**2 + (y1 - y2)**2` without any constraint on `x1`, `x2`, `y1`, and `y2`. The magnitude-of-cross-product formula contains this computation. A more careful user may define abstractions and rules for 2D points and for getting their X and Y components. This requires knowing how points are usually represented and risks more false negatives.

Example Matches. We present example matches to demonstrate how YOGO is able to find matches in the presence of paraphrases, match discontiguity, and multiple languages.

- Figure 13 gives three examples of SP2, squared distance. Figures 13a and 13c use explicit power-of-2, while 13b uses self-multiplication. These paraphrases are bridged by a Java-specific equality rule that rewrites the value of `Math.pow` function call to a generic power operation and a language-generic equality rule that rewrites `x * x` to `x ** 2`, as well as the generic rules for local variables.
- Figure 14 shows different way one can iterate over key-value entries of a map (SP7). Although this required language-specific rules to understand iterating over the key set vs the entry set of a map, we reused a lot of rules and abstractions from SP3 and SP4, which also involved maps and iteration, respectively. As a result, SP7 is fairly concise, with only 5 node patterns.

```
double distSq = Math.pow((a.getCenterX() - b.getCenterX()), 2)
     + Math.pow((a.getCenterY() - b.getCenterY()), 2);
```

(a) LitiEngine (Java)

```
return Math.sqrt(
  (p1.getX() - p2.getX()) * (p1.getX() - p2.getX())
   + (p1.getY()-p2.getY()) * (p1.getY()-p2.getY()));
```

(b) LitiEngine (Java)

```
xdistance = left.rect.centerx - right.rect.centerx
ydistance = left.rect.centery - right.rect.centery
distancesquared = xdistance ** 2 + ydistance ** 2
```

(c) PyGame (Python)

Figure 13. Excerpts that match SP2 (squared 2D distance).

```
for n, k in enumerate(self.tileset):
   s = pyglet.sprite.Sprite(self.tileset[k].image, y=y, x=x,
        batch=self.tiles_batch)
   # ...
```

(a) Cocos2D (Python)

```
for (final Map.Entry<String, List<Consumer<Float>>> entry :
     this.componentReleasedConsumer.entrySet()) {
   for (final Consumer<Float> cons : entry.getValue()) {
      pad.onReleased(entry.getKey(), cons);
   }
}
```

(b) LitiEngine (Java)

Figure 14. Excerpts that match SP7 (iterating over a map).

More discussion is available in the accompanying technical report [58], and the set of all matches is available in the accompanying artifact.

False Negatives. By extensive use of `grep`, we found several false negatives in PyGame, Cocos2D, and LitiEngine. These occur for five reasons:

- **Incompleteness of translator**: Yogo's prototype implementation translates many language constructs to the "unknown statement" node, yielding false negatives. For example, Yogo missed five instances of opening a file which used Python with-statements.
- **Incompleteness of rules**: Yogo's results depend on the rules given to it. Particularly for patterns that deal with specific APIs, missing rules cause false negatives. For example, as Yogo is currently unable to look at the Java class hierarchy, we manually created rules equating instances of several types to the `file-handler` concept. However, we missed `ImageWriter`, yielding a false positive in LitiEngine. Another notable example of missing rules: for performance reasons, Yogo does not contain rules for associativity.
- **Interprocedural code**: Yogo cannot detect patterns when the code is split across methods. For example, Yogo missed an instance of squared 2D-distance in PyGame which invoked vector-subtraction. While this limitation can be partially addressed by inlining (for a significant performance price), this is actually a fundamental limitation of PEGs.
- **Reasoning about memory**: Yogo currently contains a trivial solution to the frame problem: unless the function is marked pure, it assumes that a function may mutate all memory — and Yogo currently treats even local variables as memory lookups. Hence, Yogo may miss a match when the matching code is interleaved with a call to an impure function.
- **Timeout**: We identified one false negative due to timeout in PyGame, in a 110 SLOC function with four-level nested loops and a long if-elif chain. The function contains six time subtractions; only five are found.

More discussion of false negatives is available in the accompanying technical report [58].

5.1.2 Synthetic Fragments.

Because SP4, SP8, and SP9 occurred few times in the evaluated codebases, we supplemented with synthetic code fragments containing these patterns and near misses, designed to stress-test Yogo. We briefly describe the synthetic fragments for array-frequency count (SP4). The rest are detailed in the accompanying technical report [58].

For array frequency count, we wrote a large number of test cases in both Java and Python, including the examples of Figures 1 and 2, as well as ones wth a flipped if-else. These matched, except in the case where interleaving code contained a function call, and Yogo could not verify this call did not modify relevant state. In two more variants, the frequency counter is an array element or a field of an object rather than a simple variable. Yogo matches the example with an object-field counter, but has a false negative for the array-field counter, as its ruleset for array-index lvalues is incomplete. We also mutated several examples to produce near-misses. One notable false positive occurs because our rules for for-each loops incorrectly assume that the array is not modified during the loop.

5.1.3 Reflection on Usability.

How hard is it to write Yogo queries? Here we share our experience from this study.

For patterns which are just expressions, such as the 2D-distance formula, one can simply translate the expression into Yogo syntax, and let the Yogo engine untangle isomorphic code. It does require more thought, however, to think of relevant isomorphisms to add.

For patterns which are "procedures," such as frequency-count, there was more work to identify the charateristic subexpressions, and to identify reusable components and their respective matchers. After identifying them mentally, however, actually programming in YOGO's DSL was straightforward.

While we did not track the time spent building these queries, the lead author recalls that, even for the more complicated queries, he would finish them in under an hour.

According to one definition of design[2], programmers work by refining high-level intentions into low level code. YOGO was inspired by a tool, Sequoia [25], built to reverse this process. The work of building a YOGO query is the work of discovering the higher-level building blocks of intention.

5.2 Graal Case Study

The GraalVM [90] is a Java VM and JDK built by Oracle designed to support language-interoperability and ahead-of-time compilation. As its associated Java compiler, Graal, is itsef written in Java, the project contains 13 hand-written static analyzers, built atop Graal's own bytecode analysis infrastructure, to enforce coding guidelines and catch bugs in the project. We determined that the buggy patterns sought by 4 of these analyzers could be expressed as YOGO queries. We implemented 3 of these as YOGO queries, and verified they all caught the defects in past revisions of Graal that inspired the corresponding analyzer. But, for one of these, we realized that the YOGO query would naturally be more general than the existing analyzer, opening the possibility that YOGO could detect new bugs in the codebase. And, excitingly, this happened to be the one analyzer relevant to other codebases.

`VerifyDebugUsage` searches for several patterns akin to `Debug.log("A: "+ str)` or `Debug.log("A: \%s", node.toString)`. Code of these patterns all perform string computations whose results are discarded when debug-logging is not enabled. These are all minor performance bugs, but ones which the Graal team has aimed to eradicate from their compiler. The preferred alternative is `Debug.log("A: \%n", node)`, which does no string computations unless debug-logging is enabled.

The `VerifyDebugUsage` analyzer is 330 lines,[3] and works by manual tree-pattern matching. We immediately noticed a limitation: it would fail to detect the target defect if there was indirection through temporary variables, as in the example `str = n.toString(); Debug.log("\%s", str);` . However, YOGO by default treats such a snippet as indistinguishable from its inlined version, `Debug.log("\%s", n.toString())`.

After identifying the opportunity, to help with our deadline, we outsourced the remaining work to a programmer in India, Sreenidhi Nair. As evidence for YOGO's usablity, **in 3 days, he learned the tool well enough to implement this query**, along with queries for the other Graal checkers. The debug-usage query is 69 SLOC of YOGO DSL. Of these, 11

[3]As of SHA 4ce223a1dc

```
(deftrigger plus-string-string
 (e <- (generic/binop :+ s1 s2))
 (rules/is-string s1)
 (rules/is-string s2)
 =>
 (rules/is-string e))

(deftrigger string-alloced-argument
 (rules/obj-to-string arg)
 =>
 (rules/bad-debug-argument arg))
```

Figure 15. Extracts of query for incorrect debug usage

lines merely tag expressions as having type `String`, owing to the lack of type information in YOGO's current information, leaving 58 lines of actual query code. Figure 15 gives extracts of this code, which uses YOGO's support for auxiliary facts to identify string expressions and to mark expressions as unsuitable for use in debug-logging.

For testing, we inspected past commits which modified their checker to find instances of the bug it was designed to catch. To avoid overfitting the query, the programmer developing the query was told which commits and directories contained instances of the buggy pattern, but was not shown the instances until after successfully detecting it with YOGO.

Although Graal has a 1.2 million line codebase, as YOGO runs on each method independently, parallelizing this search was straightforward. The final run took 2.5 hours using 30 AWS instances (type `c5n.xlarge`). The search turned up many uninteresting true positives, such as defects in test code, which `VerifyDebugUsage` is not configured to check. It also turned up one example which, while an instance of the buggy pattern which should have been caught by their checker, was not an actual defect, as it was wrapped by the condition `if (log.isLoggable(Level.FINE))`. Along with these uninteresting-yet-correct matches, it also turned up one defect that could not have been found by `VerifyDebugUsage`.

Figure 16 gives the buggy code, which VerifyDebugUsage missed because of the indirection through `nodeName`. Our fix wrapped this code in a condition checking that logging was enabled, and was accepted into the Graal codebase.[4] And thus, **a 60-line YOGO query found a bug missed by a 330-line checker designed for that exact purpose**.

5.3 (Lack of) Comparison to Other Tools

As described in §6, we discovered over 70 prior tools related to semantic code search or clone detection. For most of these, the focus of the tool was too different for a meaningful comparison. We identified 4 candidates where we saw potential for an interesting experiment design, but failed for each. We ran MeCC, the clone detector based on abstract-interpretation of memory accesses, on the examples of §1

[4]https://github.com/oracle/graal/pull/1965/

```
if (object instanceof Node) {
 Node node = (Node) object;
 String loc = GraphUtil.approxSourceLocation(node);
 String name = node.toString(Verbosity.Debugger);
 if (loc != null) {
  debug.log("Context obj %s (approx. location: %s)", name, loc);
 } else {
  debug.log("Context obj %s", name);
 }
}
```

Figure 16. Defect caught by Yogo but not custom checker.

translated to C: it failed to detect similarity. Unfortunately, it only supports C, prohibiting a larger comparison. Dyclone [26] similarly only supports C. The most promising tools were the SAT-based search tool Satsy [77] and an unnamed tool that replicates it [27], but we were unable to obtain a runnable version of either tool. Beyond these 4 tools which were strong candidates for a meaningful comparison, we also substantially investigated 3 other tools (Oreo [72], DCS [18], and Alice [76]), but for each found that the problems addressed were too different, the benchmarks were a mutual poor fit, and/or any experiment protocol would have high and inescapable experimenter bias.

Although Yogo cannot be used out-of-the-box as a clone detection tool because it requires manual query definition, we also investigated BigCloneBench [78], a popular clone-detection benchmark set. We found that its inclusion criteria aligned poorly with the granular exact-matches detected by Yogo, meaning that BigCloneBench contains too many false positives and false negatives for the problem variant addressed by Yogo. We ultimately gave up on having an apples-to-apples comparison with an existing tool, and settled for a comparison based on careful literature review, presented in §6.

6 Related Work

The direct inspiration for Yogo was actually not a code search tool at all, but a program comprehension / reverse-engineering tool, Sequoia[25], which is in turn inspired by the Programmer's Apprentice [64]. Semantic Designs' Sequoia tool uses graph-pattern matching to hierarchically discover higher-level code patterns in low-level DOWTRAN code, and has been used to migrate the control software for dozens of chemical plants.[5] Sequoia proved that, with enough computational power, the insights of the Programmer's Apprentice could work at scale, and hence guided us to a code search technique based on hierarchical dataflow patterns. Its marriage to Ross Tate's E-PEGs, together with additional reasoning capabilities and careful design of a language of more abstract nodes, produced Yogo.

[5]James Koppel was an intern at Semantic Designs in 2016, though the relevant details of Sequoia have been explained at public talks

In the remainder of this section, we discuss code search. Code search has been heavily studied over the past 30 years. In this section, we provide a comprehensive survey of 70 tools for code search, and the related problems of code completion, clone-detection, and "identifying reusable components" [57, 67]. While there are countless variations of the code-search problem, they can be broadly split along two dimensions:

1. Searching the open web vs. searching within a single codebase
2. Fast searches based on superficial features vs. expensive searches based on program semantics.

While we include all categories of work in our survey, we focus on the quadrant that contains our work: semantic-search within a single codebase.

Confusingly, note that some authors use "semantic search" to refer to search based on natural language semantics rather than program semantics.

General Code Search. Classic code search is based on information retrieval techniques originally designed for textual search, which focus on the set (boolean model) or multiset (vector space model / "bag of words") of terms which appear in the document [41, 73, 89].

Many approaches try to augment these basic search techniques with additional information such as topics/categories [11, 84, 94], similarity metrics [1, 31], related terms found on StackOverflow [33, 75], the graph of which developers work on which projects [82], and search engine click-through data [60, 95]. Two interesting points in this space are Portfolio [47], which refines the initial search results by running PageRank on the call-graph, and Source Forager [30], which compares functions using a weighted average of many similarity scores, including fine-grained features such as the set of shared numeric constants.

While most works focus on improving the search results of the basic algorithms, there are several papers which study how providing a better or more-interactive UI can help code search [3, 45, 55, 76].

Outside of information-retrieval techniques, there are a few other simple query methods. Several tools offer search based on type signatures [17, 67]. There are many tools which use AST-matching either alone [56, 86] or as part of a multi-modal query [3], and at least one using a fuzzy tree-similarity algorithm [70]. An extension of these is context-matching [59], which can simultaneously match multiple subtrees and partial-subtrees. LASE [48] can similarly use multiple AST patterns.

Finding API Examples. A common variant of code search is to find a sequence of API calls that either performs a common task or returns a desired type. Typically, a query is given as a set of input types and a desired output type. Approaches use synthesis, code search, or a blend of both.

Type-directed pure-synthesis approaches include Prospector [42], CodeHint [16], and SyPet [12]. MatchMaker [93] and DemoMatch [92] use synthesis based on a a database of example traces collected by dynamic analysis. MAPO [91] is similar, but creates its database by merging static sequences of API calls.

Tools on the pure search end include Strathcona [23], XSnippet [71], and PARSEWeb [81]. All three of these are based on using heuristics to rank a set of existing API sequences, discovered in functions found by a backing general search engine. Strathcona and ParseWEB both use Google Code Search [89], while XSnippet provides its own.

Several approaches do heavier processing of identified API sequences before presenting a result to the user. SNIFF [7] uses a syntactic code-intersection procedure. Both PRIME [49] and SWIM [60] use patterns of existing API sequences to synthesize a new answer.

There are also many tools that search for API sequences as part of code completion. These tools are based on synthesis, statistical modeling, or both [4, 19, 52, 53, 62].

One unique approach is taken by RACS [38]. RACS obtains structure about special dependencies in code such as "is used in a callback." By obtaining similar information from its natural-language parser, it can handle queries such as "when someone clicks on an image, change the image source."

Deep-Learning. Several newer tools straddle the line between search based on textual or other superficial features and search based on program semantics by applying deep-learning on both natural-language and code artifacts. Approaches include running a neural network over important names [68], the program AST [88], and, for Github Semantic Code Search [21], lexer output. DCS [18], in contrast, uses more semantic features such as API-sequences. Cambronero et al [6] survey many of these techniques.

Another application of deep learning is augmenting queries based on logs of how developers reformulate queries [40].

Searching on Code Semantics. The hitherto most-common approach to matching code beyond the syntactic level has been to use program dependence graphs [13]. PDG-based tools have the common feature of being able to ignore statement ordering and interleaving with other code, but unable to deal with larger differences. Rattan et al [61] survey 72 separate clone-detection tools, finding 5 which use PDGs [15, 22, 24, 34, 36]. We further found one PDG-based tool for plagiarism detection [39] and one for code search [85].

Rattan et al identify 8 tools [8, 15, 26, 32, 34, 36, 44, 74] as focusing on semantic clone detection (often called "type-4 clones" in the literature [66]), including 3 of the PDG-based tools. Another two work by comparing the sets of identifiers [44] or API calls [8] used in a function. We discuss the other 3 [26, 32, 74] in more detail, grouped with other works using similar techniques.

Many tools are based on testing. Behavior sampling [57] and generalized behavior sampling [20] randomly run procedures based on user-supplied inputs. Reiss's system [63] searches for candidate methods using keyword and type-signature search, creates many variants of them using a small set of transformations, and then tests them. Jiang and Hu [26] built a system which can randomly test every 10-line snippet in a codebase, looking for ones that match under some permutation of variables. Their approach is based on I/O equivalence rather than semantic equivalence, meaning both snippets must produce the same outputs even on invalid inputs. They managed to run their system on the entire Linux kernel in 189 hours wall-clock time on a shared cluster.

Next are the tools based on program analysis. MeCC [32] uses a path-sensitive, interprocedural abstract interpretation to compute a summary of a procedure's memory state, and then compares functions based on the similarity of their final abstract state. It excels in finding semantic clones which read or write the same data-structure fields. However, we tested MeCC on several examples similar to Figure 1 transliterated into C, and it reported that none of them matched. Tracy [9] uses a technique based on dynamic analysis and constraint-solving to compare functions specifically in binary executables. There are also several Datalog-based program analysis tools whose query engines may be used for code search [46, 83, 87]. They are uniformly limited by the expressiveness of their underlying program analysis, which offer coarse abstractions designed for bug-finding rather than checking program equivalence.

Aside from the Datalog-based static analyzers, several other search tools use logic as their query mechanism. The works of Schügerl [74] and of Sivaraman et al [76] both collect a set of facts about procedures, and accept logical queries about these facts. Most of these facts are syntactic, and so both approaches often devolve into simultaneously searching for several AST-patterns, although Schüügerl's system has limited ability to treat procedure calls as if they were inlined, and Sivaraman's includes control-flow facts but not data-flow facts. Rollins and Wing [65] assume a world in which every function comes equipped with a logical specification, so that search can be done by matching a partial specification. Satsy [77] features perhaps the most expensive and powerful of the techniques we surveyed. Satsy takes a query as input/output examples, converts all snippets under search to SMT formulas, and then runs an SMT solver, allowing it to detect if a snippet can be specialized to satisfy the I/O examples. The downside is that it only works on a couple DSLs and a subset of Java, and requires manually-written models of library functions. It supports "most of the Java string API." We contacted the author of Satsy asking to try it, but she responded that no runnable version is available. Jiang et al [27] built a tool similar to Satsy, but extended to support loops and a few more library functions; they did not respond to our request .

Coccinelle [5, 54] is a search and rewrite tool based on control-flow-graph matching. It resembles an AST-rewriting tool, except that patterns may include the "..." wildcard, which matches an arbitrary set of control-flow paths. It supports a limited number of isomorphisms, but uses an incomplete matching procedure: it matters what order rewrite rules are given to the system. Note that, although it can rewrite matched code, it ignores the challenges of rewriting in the presence of isomorphisms: a rule to rewrite `x*2` to `x*3` will happily rewrite `2*y` to `y*3`, and, if there are overlapping matches, it can rewrite both of them, giving incorrect output.

Verified lifting [28] is the task of automatically replacing certain types of code with equivalent code in some DSL. Verified-lifting tools implement special-purpose program analyses to search for candidate lifting targets.

How do we compare? We now revisit our claim from §1, that no existing tool can detect all five array-count snippets in Figure 1 from a single query without large amounts of false positives. It is clearly true by a technicality: of the tools which use more than text and ASTs, only Github Semantic Search is for Python. That aside, there are only 7 tools to consider which (1) can deal with code equivalences beyond renaming and reordering (2) for snippets smaller than a whole function and (3) strive to return only correct results, rather than a list of results of varying relevance. Coccinelle is technically in this category, though its semantic-matching capabilities extend only slightly beyond AST-matching. Similarly, the three Datalog-based tools are only incidentally useful for code search, and only provide coarse dataflow facts. L. Jiang and Hu's testing-based approach [26] falls short because it requires that the two snippets under comparison have the exact same number of local variables. R. Jiang et al's extension to Satsy [27] is a strong contender, and may be able to recognize the four snippets which do not use `array.count()`, depending on the performance of the underlying SMT solver. From this, an easy conclusion is that recognizing the fifth snippet merely requires adding a model of `array.count()` to its library. Unfortunately, it is not possible to express counting using first-order logic [37]. Reiss [63] comes the closest, using slicing to overcome the limitation of L. Jiang and Hu. Unfortunately, the slicing (1) acts on statement-granularity, ruling out the snippet of Figure 1b when searching for array frequency-counting, and (2) depends on the return type of the query, making it inapplicable when searching for a use of the frequency-count.

We do, however, wish to point out that one of the Datalog-based tools is now the commercial product Semmle[6], which features an expansive language for writing custom static analyses. Although this is far from its intended purpose, we have not confirmed that it is not possible to write a custom program analysis to find array frequency-count functions.

In summary:

- Static-analysis based methods offer coarse abstractions not intended for code search.
- SMT-based methods have difficulty with loops and function calls.
- To preserve scalability, testing-based methods require limitations on their search for small snippets.

Equational reasoning, in contrast, can deal with arbitrary loops and functions, and detect arbitrarily small snippets.

7 Discussion, Future Work, and Conclusion

If the experiences of our first user outside the authors generalizes, YOGO and its DSL takes 3 days to learn. In return, the user gains the power to query codebases without missing matches due to paraphrase. Though we built YOGO as a code search tool, the Graal case study shows it's also usable as a bug-finder. We are also interested in exploring YOGO's use as a program comprehension tool, to hierarchically decompose arbitrary code into higher-level concepts, like the Sequoia and Programmer's Apprentice systems that inspired it. The flipside of this generality is that it's very difficult to explain exactly what kinds of problems YOGO can and cannot solve. As demonstrated in the Graal case study, users can use YOGO's term-rewriting capabilities to augment it with arbitrary computation. We have found no better description then "snippets that can be shown equivalent to a pattern using a terminating system of conditional graph-rewrite rules."

From our survey, it's safe to conclude that YOGO has the most flexible matching of any semantic code search tool. Yet our YOGO implementation is just a prototype, and there are many unexplored extensions. In addition to the challenges of rewriting discussed in §3.5, and incremental improvements to performance or reasoning, we see much opportunity for research on combining YOGO-like techniques with probabilistic matching, learning of rules (e.g.: [79]), and matching/unification modulo theories [29].

Like many SMT-based tools before it, YOGO is part of a larger trend of taking heavyweight reasoning techniques built for compilers and verification, and scaling them to a software-engineering context. We look forward to seeing more work in this area to bring about the age of smart tools.

Acknowledgments

We give a large thank you to Sreenidhi Nair, the research programmer who did the query development and devops work for the Graal case study, and also assisted us packaging the artifact. Thanks to Yaniv David, Soya Park, and José Cambronero for general advice on code-search research. Thanks to Zenna Tavares, Zoe Anderson, and Ross Tate for comments on earlier drafts. We also thank the anonymous reviewers, and our shepherd, Mukund Raghothaman, for their help. Thanks to Dr. Thomas Würthinger for pointing us to the Graal checkers.

[6]http://semmle.com

References

[1] Sushil Krishna Bajracharya, Joel Ossher, and Cristina Videira Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010*. 157–166.

[2] Don S. Batory, Rui C. Gonçalves, Bryan Marker, and Janet Siegmund. 2013. Dark Knowledge and Graph Grammars in Automated Software Design. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. 1–18. https://doi.org/10.1007/978-3-319-02654-1_1

[3] Andrew Begel. 2007. Codifier: A Programmer-Centric Search User Interface. In *Proceedings of the Workshop on Human-Computer Interaction and Information Retrieval*. 23–24.

[4] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2933–2942.

[5] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2009. A Foundation for Flow-Based Program Matching: Using Temporal Logic and Model Checking. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 114–126.

[6] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 964–974.

[7] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*. York, UK, 385–400. https://parlab.eecs.berkeley.edu/sites/all/parlab/files/sniff{_}0.pdfhttp://link.springer.com/10.1007/978-3-642-00593-0{_}26

[8] Seokwoo Choi, Heewan Park, Hyun-il Lim, and Taisook Han. 2009. A static API birthmark for Windows binary executables. *Journal of Systems and Software* 82, 5 (2009), 862–873.

[9] Yaniv David and Eran Yahav. 2014. Tracelet-Based Code Search in Executables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*. 349–360.

[10] David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (2005), 365–743. https://www.hpl.hp.com/techreports/2003/HPL-2003-148.pdf

[11] Frederico Araújo Durão, Taciana A. Vanderlei, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2008. Applying a Semantic Layer in a Source Code Search Tool. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*. 1151–1157.

[12] 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, Paris, France, January 18-20, 2017*. 599–612.

[13] Jeanne Ferrante and Joe D Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.

[14] Charles L Forgy. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19 (1982), 17–37. http://www.csl.sri.com/users/mwfong/Technical/RETEMatchAlgorithm-ForgyOCR.pdf

[15] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. 321–330.

[16] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 653–663.

[17] Isabel Garcia-Contreras, José F. Morales, and Manuel V. Hermenegildo. 2016. Semantic Code Browsing. *TPLP* 16, 5-6 (2016), 721–737.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*. 933–944.

[19] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 27–38.

[20] Robert J. Hall. 1993. Generalized Behavior-Based Retrieval. In *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*. 371–380.

[21] Hamel Husain and Ho-Hsiang Wu. 2018. Towards Natural Language Semantic Code Search. https://github.blog/2018-09-18-towards-natural-language-semantic-code-search/.

[22] Yoshiki Higo and Shinji Kusumoto. 2011. Code Clone Detection on Specialized PDGs with Heuristics. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. 75–84.

[23] Reid Holmes and Gail C. Murphy. 2005. Using Structural Context to Recommend Source Code Examples. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. 117–125.

[24] Susan Horwitz. 1990. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*. 234–245.

[25] Ira Baxter and Randall Matthias. 2017. Dow Chemical awards contract extension to Semantic Designs for Process Control Software Reengineering. http://www.semdesigns.com/Announce/DOW_PRWebRelease_May2017_14309535.pdf.

[26] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 81–92.

[27] Renhe Jiang, Zhengzhao Chen, Zejun Zhang, Yu Pei, Minxue Pan, and Tian Zhang. 2018. Semantics-Based Code Search Using Input/Output Examples. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. 92–102.

[28] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. 711–726.

[29] Deepak Kapur and Paliath Narendran. 1987. Matching, Unification and Complexity. *ACM SIGSAM Bulletin* 21, 4 (1987), 6–9.

[30] 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). arXiv:1706.02769 http://arxiv.org/abs/1706.02769

[31] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *36th International Conference on Software Engineering, ICSE '14*. 664–675.

[32] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwangkeun Yi. 2011. MeCC: Memory Comparison-based Clone Detector. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 301–310.

[33] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY - A Code-to-Code

Search Engine. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM Press, New York, New York, USA, 946–957.

[34] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*. 40–56.

[35] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 122.

[36] Jens Krinke and Lehrstuhl Softwaresysteme. 2001. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering*. Washington, DC.

[37] Dietrich Kuske and Nicole Schweikardt. 2017. First-Order Logic with Counting. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. 1–12.

[38] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-Aware Code Search for JavaScript Frameworks. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*. 690–701.

[39] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by program Dependence Graph Analysis. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*. 872–881.

[40] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural Query Expansion for Code Search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. ACM, New York, NY, USA, 29–37.

[41] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *30th IEEE/ACM International Conference on Automated Software Engineering*. 260–270.

[42] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 48–61.

[43] Panagiotis Manolios. 2001. Mechanical Verification of Reactive Systems. (2001).

[44] Andrian Marcus and Jonathan I. Maletic. 2001. Identification of High-Level Concept Clones in Source Code. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*. 107–114.

[45] Lee Martie, André van der Hoek, and Thomas Kwak. 2017. Understanding the Impact of Support for Iteration on Code Search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 774–785.

[46] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. 365–383.

[47] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*. 111–120.

[48] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 502–511. https://doi.org/10.1109/ICSE.2013.6606596

[49] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-Based Semantic Code Search over Partial Programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012,*. 997–1016.

[50] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (1979), 245–257.

[51] Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (1980), 356–364.

[52] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-Based Statistical Language Model for Code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 858–868.

[53] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2012. Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 69–79.

[54] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. 2007. SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers. *Electr. Notes Theor. Comput. Sci.* 166 (2007), 47–62.

[55] Soya Park, Amy X. Zhang, and David R. Karger. 2018. Post-literate Programming: Linking Discussion and Code in Software Development Teams. In *The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings, UIST 2018, Berlin, Germany, October 14-17, 2018*. 51–53.

[56] Santanu Paul and Atul Prakash. 1994. A Framework for Source Code Search Using Program Patterns. *IEEE Trans. Software Eng.* 20, 6 (1994), 463–475.

[57] Andy Podgurski and Lynn Pierce. 1992. Behavior Sampling: A Technique for Automated Retrieval of Reusable Components. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992*. 349–360.

[58] Varot Premtoon. 2019. *Multi-Language Code Search*. Ph.D. Dissertation. Massachusetts Institute of Technology. http://doi.org/handle/1721.1/7582

[59] Cosmin A Radoi. 2018. *Toward Automatic Programming*. Ph.D. Dissertation. Ph. D. thesis, University of Illinois at Urbana-Champaign.

[60] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 357–367.

[61] Dhavleesh Rattan, Rajesh Kumar Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information & Software Technology* 55, 7 (2013), 1165–1199.

[62] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 419–428.

[63] Steven P. Reiss. 2009. Semantics-Based Code Search. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 243–253.

[64] Charles Rich and Richard C Waters. 1988. The Programmer's Apprentice. *Computer* 21j, 11 (1988), 10–25.

[65] Eugene J. Rollins and Jeannette M. Wing. 1991. Specifications as Search Keys for Software Libraries. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*. 173–187.

[66] Chanchal Kumar Roy and James R Cordy. 2007. A Survey on Software Clone Detection Research. *QueenâĂŹs School of Computing TR* 541, 115 (2007), 64–68.

[67] Colin Runciman and Ian Toyn. 1989. Retrieving Re-Usable Software Components by Polymorphic Type. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989*. 166–173.

[68] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018*. 31–41.

[69] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian G. Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 191–201.

[70] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. 2006. Detecting Similar Java Classes Using Tree Algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. 65–71.

[71] Naiyana Sahavechaphan and Kajal T. Claypool. 2006. XSnippet: mining For sample code. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 413–430.

[72] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 354–365. https://doi.org/10.1145/3236024.3236026

[73] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. 1157–1168.

[74] Philipp Schügerl. 2011. Scalable Clone Detection Using Description Logic. In *Proceeding of the 5th ICSE International Workshop on Software Clones, IWSC 2011, Waikiki, Honolulu, HI, USA, May 23, 2011*. 47–53.

[75] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 945.

[76] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active Inductive Logic Programming for Code Search. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 292–303.

[77] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the Search for Source Code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 26.

[78] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating Clone Detection Tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. 131–140. https://doi.org/10.1109/ICSM.2015.7332459

[79] Ross Tate, Michael Stepp, and Sorin Lerner. 2010. Generating Compiler Optimizations from Proofs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 389–402. https://doi.org/10.1145/1706299.1706345

[80] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the Symposium on Principles of Programming Languages*. Savannah, GA.

[81] Suresh Thummalapenta and Tao Xie. 2007. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 204–213.

[82] Rahul Venkataramani, Allahbaksh M. Asadullah, Vasudev D. Bhat, and Basavaraju Muddu. 2013. Latent Co-development Analysis Based Semantic Search for Large Code Repositories. In *2013 IEEE International Conference on Software Maintenance*. 372–375.

[83] Mathieu Verbaere, Elnar Hajiyev, and Oege de Moor. 2007. Improve Software Quality with SemmleCode: An Eclipse Plugin for Semantic Code Search. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 880–881.

[84] Shaowei Wang, David Lo, and Lingxiao Jiang. 2011. Code Search via Topic-Enriched Dependence Graph Matching. In *18th Working Conference on Reverse Engineering, WCRE 2011*. 119–123.

[85] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. 2010. Matching Dependence-Related Queries in the System Dependence Graph. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. 457–466.

[86] Louis Wasserman. 2013. Scalable, Example-Based Refactorings with Refaster. In *Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013*. 25–28.

[87] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. 131–144.

[88] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. 87–98.

[89] Wikipedia. 2019. Google Code Search — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Google%20Code%20Search&oldid=904263675. [Online; accessed 24-August-2019].

[90] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 187–204. https://doi.org/10.1145/2509578.2509581

[91] Tao Xie and Jian Pei. 2006. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. 54–57. https://doi.org/10.1145/1137983.1137997

[92] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API Discovery from Demonstrations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 64–78.

[93] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. 2011. Data-Driven Synthesis for Object-Oriented Frameworks. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011,*. 65–82.

[94] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. 2018. Expanding Queries for Code Search Using Semantically Related API Class-names. *IEEE Trans. Software Eng.* 44, 11 (2018), 1070–1082. https://doi.org/10.1109/TSE.2017.2750682

[95] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing Developer Assistant: Improving Developer Productivity by Recommending Sample Code. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 956–961.