Mining Idioms in the Wild

Aishwarya Sivaraman*
UCLA
USA

Rui Abreu, Andrew Scott,
Tobi Akomolede, Satish Chandra
Meta Platforms, Inc.
USA

ABSTRACT

Existing code repositories contain numerous instances of code patterns that are idiomatic ways of accomplishing a particular programming task. Sometimes, the programming language in use supports specific operators or APIs that can express the same idiomatic imperative code much more succinctly. However, those code patterns linger in repositories because the developers may be unaware of the new APIs or have not gotten around to them. Detection of idiomatic code can also point to the *need* for new APIs.

We share our experiences in mining imperative idiomatic patterns from the Hack repo at Facebook. We found that existing techniques either cannot identify meaningful patterns from syntax trees or require test-suite-based dynamic analysis to incorporate semantic properties to mine useful patterns. The key insight of the approach proposed in this paper — *Jezero* — is that semantic idioms from a large codebase can be learned from *canonicalized* dataflow trees. We propose a scalable, lightweight static analysis-based approach to construct such a tree that is well suited to mine semantic idioms using nonparametric Bayesian methods.

Our experiments with *Jezero* on Hack code show a clear advantage of adding canonicalized dataflow information to ASTs: *Jezero* was significantly more effective in finding new refactoring opportunities from unannotated legacy code than a baseline that did not have the dataflow augmentation.

1 INTRODUCTION

An idiom is a syntactic fragment that frequently recurs across software projects. Idiomatic code is usually the most natural way to express a certain computation, which explains its frequent recurrence in code. An idiomatic imperative code fragment often has a single semantic purpose that, in principle, can be replaced with API calls or functional operators.

To illustrate the motivation for this work, consider the imperative code examples in the Hack programming language[1] in Figure 1, (a), (c) and (e). These examples—adapted from the codebase at Facebook—loop over a vector and accumulate some value in the loop body. To capture this idiom, Hack supports a more functional `Vec\map_with_key` API, and we do find instances where a developer refactored code to replace a loop with a map call; for instance, see (b), (d), and (f). This kind of refactoring is not unique to Hack; examples in other programming languages abound, such as using LINQ APIs in C# [1] or Python's list comprehensions.

Why do imperative idioms continue to linger in code? This can be attributed to: (1) developers being unaware of the API that can replace the imperative code, or (2) a new API construct being introduced and imperative locations not being updated consistently to use this construct, or (3) an API that would simplify this idiomatic pattern has not been included in the language yet. Identifying such imperative idiomatic patterns and replacing it with corresponding API calls or operators can help in maintainability and comprehensibility of the codebase. Moreover, within Facebook, an approach that identifies likely idiomatic code location would serve as an educational tool for developers in an IDE or when submitting a pull request. Besides, identification of common idioms may provide data-driven guidance to language developers for new language constructs (this purpose is outside this paper's scope).

1.1 Finding missed refactoring opportunities

Suppose we want a tool that looks at past instances of refactorings to learn imperative idiomatic patterns to identify additional opportunities of similar refactorings either in existing or new code. For example, the code in Figure 1a, 1c, and 1e was replaced, respectively, with the code in Figure 1b, 1d, and 1f, where each of the examples was rewritten using the map API. We want the tool to learn a general pattern from Figure 1a, 1c, and 1e which when exists in the code, begs for refactoring. A more powerful tool would also suggest the refactored code that is drop-in replacement for the existing code, but the design of semantically correct code transformation is a separate hard problem, outside the scope of this work.

At first blush, this may look like a pattern matching or clone detection problem, where a code fragment that is a candidate for refactoring might be a clone of code that *was* refactored in the past to introduce an API call. Another candidate approach might aim to extract generalizable code transformations from a small set of specific examples of transformations [3, 8, 17, 23].

However, identifying the pattern for `Vec\map_with_key` API—let alone the transformation—from the examples in Figure 1 is non-trivial for the following reasons:

(1) Code that maps values to an accumulator may have different types. For example, in Figure 1a we have string accumulation whereas in Figure 1c we have a vector accumulation. Therefore, any tool that can identify this pattern would need to identify the semantic pattern that each of the example in Figure 1 is accumulating to a collection variable.

*Work performed as an intern at Meta Platforms, Inc., Menlo Park.
[1]Hack is a programming language for the HipHop Virtual Machine, created by Facebook as a dialect of PHP: https://hacklang.org/.

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05.
https://doi.org/10.1145/3510457.3513046

```
$output = '';
foreach ($outputs as $key => $value) {
  $output .= self::getOutput(
    $key, $value,
    $task, $pair, ); }
```

(a) Example 1 — Imperative Version

```
$output .= Str\join(
    Vec\map_with_key(
        $outputs,
        ($key, $value) ==> {
            return self::getOutput(
                $key, $value,
                $task, $pair,); }, ), '',);
```

(b) Example 1 — API Version

```
$call_stack_nodes = vec[];
foreach ($identifier_to_id as $identifier => $id) {
 $call_stack_nodes[] = shape(
   'id' => $id,
   'vertex' => $nodes[$id]
   'tally' => $identifier_to_count[$identifier],
   'fraction' =>
     $this->fraction($identifier_to_count[$identifier],
                  $total_count), ); }
```

(c) Example 2 — Imperative Version

```
$call_stack_nodes = Vec\map_with_key(
    $identifier_to_id,
    ($identifier, $id) ==> shape(
        'id' => $id,
        'vertex' => $nodes[$id],
        'tally' => $identifier_to_count[$identifier]
        'fraction' =>
            $this->fraction($identifier_to_count[$identifier],
             $total_count), ); );
```

(d) Example 2 — API Version

```
$versions = vec[];
foreach ($ids as $key => $value) {
  $versions[] = tuple($value, $names[$key]); }
```

(e) Example 3 — Imperative Version

```
$versions = Vec\map_with_key(
    $ids,
    ($key, $value) ==> tuple($value, $names[$key]); );
```

(f) Example 3 — API Version

Figure 1: Examples of idiomatic imperative code and its corresponding version rewritten with map API.

(2) Code may be interleaved with other code, as in Figure 1c. In this example, although the accumulation is to a vector variable, they have additional code that operates on functions and variables other than the key and value. Additionally, code in Figure 1a interleaves iteration and string concatenation.

Therefore, a naive syntactic-pattern-based approach would fail to identify common patterns that matches all examples in Figure 1. A different potential solution is to define a domain-specific language and let developers handcraft custom rules to identify semantic patterns. This requires significant manual effort, and less experienced programmers might not know how to generalize these patterns.

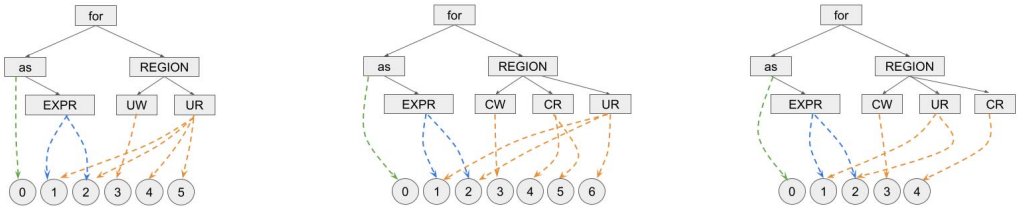
We turn to the statistical pattern mining work by Allamanis and Sutton [2], which has shown the possibility of finding patterns in abstract syntax trees (ASTs) that correspond to idioms based on their frequency of occurrence. They use probabilistic tree substitution grammars (pTSG), a non-parametric Bayesian machine learning technique to find idioms (we give an overview of this technique in Section 3.2). While this is an exciting idea, in practice this does not work very well out of the box, as Allamanis et al. report in their follow-up work [1] (and as we found as well, see Section 4.) Because purely syntactic idioms are oblivious to semantics, they capture only shallow patterns that are not useful for our end purpose.

In subsequent work, Allamanis *et al.* [1] propose to use ASTs augmented with variable mutability and function purity information (see overview in Section 2.) They found that this worked well for identifying idiomatic loops across 11 projects with 5,548 methods. Unfortunately, we found practical issues with the enhancement in [1]: (1) the technique in [1] requires annotations (which requires additional developer effort) or dynamic analysis (which requires test cases to exercise specific portions of code) to infer those, neither of which were an option for us; (2) it can only match patterns with exact lexical order of appearance of variables, and (3) it cannot detect patterns interleaved with other code. Based on our pencil and paper simulation with known mutability and purity information, their approach fails to learn a pattern that matches the examples in Figure 1 (see Section 2 for further discussion).

We propose *Jezero*, an approach that works around the practical complexities of the work of Allamanis *et al.* [1]. Our approach adds semantic information to ASTs in a different way: approximate dataflow information represented as an extension of the AST. *Jezero* automatically learns semantic patterns from a large codebase over tree structures–dataflow augmented ASTs–that are generated leveraging a cheap, syntactic dataflow analysis. Our key insight is that semantic patterns can be captured as *canonical* dataflow trees. This observation is inspired by the seminal Programmer's Apprentice paper [22] idea that high-level concepts can be identified as dataflow patterns. In fact, recent works in the area of code search [19], code clone detection [28], and refactoring [15] also use this insight and use dataflow analysis to identify semantically similar code.

For instance, a desirable dataflow pattern that summarizes the examples in Figure 1 is: `foreach` contains a `datawrite` to a `collection` variable with `dataread` from first and second `primitive` variables and the first `collection` variable in the order of their definition. To learn such a pattern, we collect approximate dataflow information from their abstract syntax trees and construct a dataset of canonicalized trees as described in Section 3.1. We then mine for dataflow patterns using a non-parametric Bayesian ML technique (see Section 3.2) [2], to which, our representation looks just like any other tree. Figure 5 shows the tree pattern that *Jezero* mined and that matches all three examples of Figure 1.[2]

[2]Due to the statistical process, the patterns mined sometimes may not cover all relevant aspects of the desirable pattern.



(a) Example 1 **(b) Example 2** **(c) Example 3**

Figure 2: Coiled ASTs for the imperative code in Figure 1: the basis for computing semantic idioms in Allamanis *et al.* [1]. The variables map to references illustrated in circles. (The numbering is unique per example, not shared.)

1.2 Contributions

This paper makes the following contributions:

- We present a new canonicalized tree representation based on inexpensive dataflow to mine semantic idioms. The approach takes as input a code corpus, generates a tree for each method augmented with dataflow information, and similar to [2], uses Bayesian learning methods to mine idiomatic patterns. Our tree representation overcomes the practical problems in adopting the closest previous work [1].
- We present *Jezero* a tool that implements both idiom learning and identification of refactoring opportunities that works at the scale of Facebook code base.
- We evaluate *Jezero* on the task of mining idioms for loopy map/filter code. The mining is done over 1347 (refactoring) instances per API taken from Facebook's Hack codebase. On an evaluation set, we found *Jezero*'s F1 score to be 0.57, significantly better than a baseline technique with 0.08.
- We evaluated *Jezero* for identifying new, hitherto unknown opportunities for refactoring code to introduce APIs. Using the top-ranked idioms, we found 807 matches against the Facebook code base containing 13770 Hack methods; the average precision of finding real opportunities was 0.60. The baseline, without dataflow, matched a mere 23 locations.
- We also conducted an initial informal user study with 20 developers who confirmed the usefulness of *Jezero* to suggest potential new refactoring locations.

To our knowledge, ***Jezero* is unique in its ability to find refactoring opportunities from legacy code, based on purely unsupervised learning, and without requiring annotations or dynamic analysis.** Moreover, we expect the ideas in *Jezero* to carry over to other languages such as Python, which over time have provided more succinct ways to express idiomatic code.

Intended Use. *Jezero* looks at past instances of refactorings and then identifies additional opportunities for similar refactorings in existing or new code; it does *not* propose transformations by itself. We intend *Jezero* to be used as a tool to point out missed idiomatic usage when submitting newly-authored code, or when undertaking mass code cleanup exercises. We show, empirically, that finding promising opportunities for refactoring is indeed feasible.

2 BACKGROUND ON MINING IDIOMS

Allamanis and Sutton [2] have addressed the problem of idiom mining as an unsupervised learning problem and proposed a probabilistic tree substitution grammar (pTSG) inference to mine idiomatic patterns. In this work, they mine syntactic idioms from ASTs; however, in their following work [1] they show that syntactic idioms tend to capture shallow, uninterpretable patterns and fail to capture widely used idioms. Data sparsity and extreme code variability are cited as the reasons for shallow idioms. Therefore, to mine interesting idioms and to avoid sparsity, the authors introduce semantic idioms. Semantic idioms improve upon syntactic idioms through a *coiling* process [1]. Coiling is a graph transformation that augments standard ASTs with semantic information to yield coiled ASTs (CASTs). These CASTs are then mined using probabilistic tree substitution grammars (pTSG), a machine learning technique for finding salient (and not just frequent) patterns [5]. They infer semantic properties such as variable mutability and function purity using a testing-based analysis. For the libraries that do not have test suites, the authors manually annotate with the required properties. The lower path in Fig 3 shows the overall process.

Using the semantic information, the coiling rewrite phase augments the nodes with variable mutability and distinguishes collections from other variables. The pruning phase retains only subtrees rooted at loop headers and abstracts expressions and control-free statement sequences to *regions* to reduce sparsity. Specifically, they abstract loop expressions into a single `EXPR` node, labeled with variable references. They use `REGION` nodes to encode purity of variables; the purity node types include read (R), write (W), and read-write (RW), and these nodes further differentiate between primitive (prefixed by U) and collection (prefixed by C) variables. Note that region nodes in this work only consider variable mutability, i.e., whether variables, being it collection or primitive, are read from or written to. While this representation effectively captures a class of semantic idioms, it is not sufficient to capture refactoring idioms that require additional flow information between variables. Figure 2 shows the CASTs for the examples in Figure 1.

Despite the effectiveness of the proposed methods, they suffer from limitations that prohibit direct application for identifying code patterns as in Figure 1 — which do occur in realistic and large codebases. Specifically,

(1) To construct `REGION` nodes, prior work relied on manual annotations or testing-based dynamic analysis. Both these efforts are expensive and might not be available for all codebases. Certainly, this is not available in legacy codebases.

(2) The augmented trees contain variable references which are numbered based on their lexicographical ordering. Hence, two loops with the same semantic concept but with a different number of variable references will have different patterns. Looking at Figure 2, at this level of abstraction, it is not clear that these trees are about the same idiom.
(3) Further, due to the lexicographical ordering, loops with the same concept but with additional interleaved code statements will most likely have different patterns.

Following these limitations, despite the code in Figure 1 being arguably about very similar constructs, Allamanis *et al.*'s approach [1] would fail to consider the examples as being part of the same idiom. The desired idioms, shown in Figure 2, albeit similar, are sufficiently distinct (e.g., ordering of variables) to be considered the same.

While this is the state-of-the-art in idiom mining, the fact that it requires dynamic analysis makes it impractical to be used in our codebase. As such, in Section 4, we will instead compare our approach with the AST-based tree representation for idiom mining proposed by Allamanis *et al.* [2].

3 PROPOSED APPROACH: *JEZERO*

In this work, we propose a new canonicalized dataflow tree representation that overcomes the limitations of prior work listed in Section 2. The upper path of Figure 3 provides an overview of our tool *Jezero*; as is clear, the difference with [1] is that we eschew coiling, and instead work with dataflow augmented trees. Section 3.1 describes the construction of dataflow augmented trees, which is our new technical contribution, and Section 3.2 gives an overview of the unsupervised idiom learning and sampling approach, which is the same as in previous work [2]. Note that [2] goes directly from code ASTs to pTSG, without any tree augmentation.

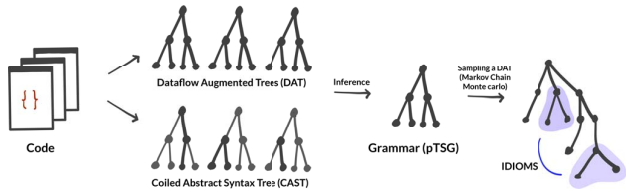


Figure 3: Overview of the steps in mining-based approaches. (This paper only uses DAT, not CAST.)

Using *Jezero* involves the following steps: (1) provide a corpus that may contain instances of the idiomatic pattern to be uncovered; (2) let *Jezero* mine the patterns and come up with a most suitable one(s) using its ranking heuristics; and (3) use *Jezero* to point out new code locations where similar refactoring can be carried out.

3.1 Dataflow Augmented Trees

The key insight of *Jezero* is that high-level concepts can be identified as dataflow patterns. Furthermore, these patterns can be captured and represented as canonical trees using an inexpensive dataflow analysis procedure. The problem with representing dataflow information in detail is that there is not enough commonality across specific dataflow graphs for a useful semantic pattern to emerge using a statistical process of mining patterns. This challenge is often referred to as the *sparsity* problem [2].

Jezero combats the sparsity problem with an abstraction that relies on dataflow information structured in a canonicalized way to capture high-level semantic concepts. To construct a dataflow augmented tree; first, we extract approximate type information from the AST. Next, we propose a lightweight algorithm that uses the extracted types and information present in the AST to derive flow information as dataflow tables. To mine useful idiomatic patterns, we propose a new tree representation that is amenable to the unsupervised mining algorithm. This canonicalized tree is constructed using information from the dataflow tables. Additionally, we make the following assumptions: (1) mining trees at method level is sufficient to capture refactoring idioms, (2) it is sufficient to keep the control flow structure and collapse control-free sequence code to a region (an approximation that is often used in static analyses [24]), and (3) the side effects, if any, of function calls are inconsequential to the dataflow information that we intend to capture.

Static Extraction of Type Information. To encode semantic information, we need to capture the data *type* of each variable. However, precise type information of the variables is often not necessary and can be counterproductive. For example, the type of variable `output` in Figure 1a is `string` whereas the type of variable `call_stack_nodes` in Figure 1c is `vector`. Therefore, having precise type information would lead to different patterns. Whereas, if the role of those variables in both cases is to act as a collection, we want to only ascribe a `collection` type to both.

We overcome the need for an expensive analysis algorithm by proposing an approximate type analysis based on information available in the ASTs. In our approach, variables are assigned as either `collection`, `object` or `primitive` type. Each variable is assigned `primitive` as the default type and, based on hints from the syntax tree, the type may be modified to `collection` or `object`. For example, if a variable contains a subscript operator, it is assigned a `collection` type. Similarly, if a variable contains the arrow operator (or equivalent operators in other programming languages) it is assigned an `object` type. At the end of this procedure, we have a type table that assigns types for each variable in the method.

As an example, Table 1 summarizes the types for the code example in Figure 1c. Note that these types are particularly useful for identifying map/filter APIs, and can be tweaked when looking for other API-related patterns. For instance, *string* types would be necessary when searching for patterns using string APIs.

Variable	Type
call_stack_nodes	collection
identifier_to_id	collection
identifier	primitive
id	primitive
nodes	collection
identifier_to_count	collection
this	object
total_count	primitive

Table 1: Inferred types for the variables in Figure 1c.

Static Extraction of Dataflow Information. We use dataflow tables to capture data writes and data reads that happen in a code block. To construct these tables, we propose a lightweight analysis that derives dataflow information based on the AST and the type table collected in the previous step. We mention at the very outset that this dataflow representation is not intended to be *sound*, as needed in compiler optimizations. This choice lets us get away with specific choices that are effective for the purpose at hand. The analysis computes dataflow table σ at each control-flow point like `if`, `foreach`, etc. σ encodes the data reads that a variable depends on. Formally, $\sigma \in Ref \rightarrow 2^{Ref}$, where Ref is a tuple containing a canonicalized identifier (id) and a variable being referenced.

Identifier for a data write is generated using the variable type and the order of appearance of the write in the current control-flow block. In case of Figure 1c, the unique identifier for variable *call_stack_nodes* would be *collection_write_0* since it is the first collection variable being written to, although it is the second collection variable in the order of appearance. $\mathcal{R}$ represents a set of *read references*. Identifier for a data read is generated using the variable type and the order of appearance in the control-flow block. For the example, in Figure 1c, read reference for the variable *id* would be $(primitive\_1, id)$. Examples of flow operations, $f[\![\cdot]\!]$, include:

$$f[\![x := y\ op\ z]\!] = [Ref_x \rightarrow \{\mathcal{R}_y \cup \mathcal{R}_z\}]\sigma$$
$$f[\![x := fc(y, z, k)]\!] = [Ref_x \rightarrow \{\mathcal{R}_y \cup \mathcal{R}_z \cup \mathcal{R}_k\}]\sigma$$
$$f[\![foreach(x\ as\ y \implies z)\{\}]\!] = [Ref_y \rightarrow \mathcal{R}_x; Ref_z \rightarrow \mathcal{R}_x]\sigma$$

The first flow function calculates σ when there is an assignment of an expression to a variable. We compute a unique reference Ref_x and compute *read references* for each variable in the expression. We then take a union of these references to update the dataflow table; Ref_x now maps to these set of read references. The second flow function is for assignment of the return values of a function call to a variable. We compute an unique identifier and read references similar to the previous flow function. The third flow function is for a `foreach` statement with an empty body. Since we have data writes to two variables (y and z) we create two unique references Ref_y and Ref_z. We further compute the read references for each of these variables and update σ. For a given data write, we identify all read dependencies using a fix-point computation. Table 2 illustrates the state of σ in two iterations of the fix-point computation in Figure 1c.

A key aspect of this representation of dataflow tables — essential to overcome the limitations of the approaches detailed in Section 2 — is the fact that we propose a new canonicalized label (id) for each dataflow operation. Each label is obtained by concatenating the data type of the variable with a number. This canonicalized label helps overcome the lexical ordering issues of previous approaches. In particular, we propose that each type of data write has its own numbering. For example, primitive writes have their own numbering which is incremented whenever there is a data write to a primitive variable. This canonicalization allows for interleaved data writes to different types of variables. While the data writes have special numbering, the data read references are computed based on their order of appearance. Hence, the same variable can have a different data read and write reference. Nested control-flow blocks require construction of σ to take into account the direction of information flow. There are two choices regarding this flow (1) top-down, where σ is carried from the outer to the inner code block (2) bottom-up, where σ is carried from the inner to the outer code block. The choice of information flow influences the type of idioms we mine. Top-down flow allows to capture idioms that require context information. For instance, consider Listing 1 where a collection variable `results` in the inner loop is populated with the result of function calls on the items in the `values` collection.

Data Write	Data Read
(primitive_write_0, identifier)	(collection_0, identifier_to_id)
(primitive_write_1, id)	(collection_0,identifier_to_id)
(collection_write_0, call_stack_nodes)	(primitive_0, identifier), (primitive_1, id), (collection_2,nodes), (collection_3, identifier_to_count), (object_0, this), (primitive_2, total_count)

(a) First Iteration

Data Write	Data Read
(primitive_write_0, identifier)	(collection_0, identifier_to_id)
(primitive_write_1, id)	(collection_0,identifier_to_id)
(collection_write_0, call_stack_nodes)	**(collection_0, identifier_to_id)** (primitive_0, identifier), (primitive_1, id), (collection_2,nodes), (collection_3, identifier_to_count), (object_0, this), (primitive_2, total_count)

(b) Second Iteration.

Table 2: Figure 1c's Intermediate Dataflow tables (σ).

```
foreach ($identifiers as $key => $values) {
    $exp = self::computeExp($key);
    $results = self::computeFirstSecond($key);
    foreach ($values as $item) {
      $results[]= self::computeResult($item, $exp);
    }
}
```

Listing 1: Nested control-flow block example.

We cannot directly assign the result of `Vec\map_with_key` to `results`, rather we have to `Vec\concat` with the items already in `results`. While the top-down information flow can lead to richer idioms, it suffers from two problems: (1) tree sparsity, since no two code blocks share the almost similar context information (2) the learning algorithm proposed by [1] learns context free grammars; however, if we want to use context information, a mining approach that can learn context *sensitive* grammars is needed.

On the other hand, bottom-up information flow allows us to identify local patterns which avoids the sparsity problem and is amenable to the learning algorithm. Hence, in *Jezero*'s σ construction, information flow is bottom-up, i.e., from inner to outer code block. Here, for each control flow node, we recursively compute dataflow table for inner block and update the outer block using ψ - a merge operator. The operator takes as input two dataflow tables and returns a merged dataflow table, formally, $\psi : \sigma_o \rightarrow \sigma_i \rightarrow \sigma_m$, where σ_o and σ_i represent the outer and inner-block respectively;

and σ_m is the merged table. The update rules for ψ are:

$$(x, r) \in \sigma_o \wedge (x, \_) \notin \sigma_i \implies [x \rightarrow r]\sigma_m$$
$$(x, \_) \notin \sigma_o \wedge (x, \_) \in \sigma_i \implies \sigma_m$$
$$(x, r1) \in \sigma_o \wedge (x, r2) \in \sigma_i \implies [x \rightarrow$$
$$fix(r1 \cup \{(id_o, v) | (id, v) \in r2 \wedge v \in \sigma_o\})]\sigma_m$$

Table 3 illustrates the working of the merge operator for the example in Listing 1. *Jezero* starts by computing the information for each inner control flow block. Table 3b illustrates the the dataflow table (σ_i) for the inner `foreach` loop. Table 3c computes the partial dataflow table (σ_o) for the outer block without the nested loop. Now we carry out merge using the rules of ψ operator. We retain the first three entries of σ_o as per the first rule of ψ. Next, we discard the first row of σ_i as per the second rule of ψ. For write to *results* variable, we need to merge and update the dataflow from the inner block. We use the third rule of ψ to collect all read references whose variables are declared in the outer context and update the identifier to reflect their position in the outer code block (see Table 3c).

Despite the fact that this bottom-up-only dataflow computation is incomplete, and that traditional context-sensitive analysis may provide sound semantic information, we make this choice purposely as it works well for finding idioms.

Tree Representation. The pattern mining algorithm we use (see Section 3.2) learns tree fragments given a context free grammar. Therefore, having as a starting point the dataflow information captured as tables, we need a suitable tree representation that is amenable to (tree) pattern mining. In this work, we replace a control-free sequence of statements with trees that capture dataflow information. This tree contains information about the data writes and data reads that happen in a code block. To ensure that these trees are compatible with the underlying learning technique, we propose the following canonicalized tree representation.

To differentiate between distinct data writes, our proposed tree representation always contains a set of (distinct) child nodes that represent data writes to `collection`, `primitive`, or `object` type. This design choice helps in overcoming the limitation with different number of write statements, since the learning algorithm will learn to retain only the common data write pattern (i.e., the common subtree). Additionally, to account for the dataflow in the loop header, we add primitive write dataflow nodes which shows the flow from the collection being iterated over to the loop header variables. *Jezero* models dataflow tables as trees using the following grammar:

$$\begin{aligned}
\langle \text{region} \rangle &\models \langle primitive_write \rangle \\
&\quad \langle collection_write \rangle \langle object_write \rangle \\
\langle primitive_write \rangle &\models (\langle write_region \rangle \langle primitive_write \rangle)* \\
\langle collection_write \rangle &\models (\langle write_region \rangle \langle collection_write \rangle)* \\
\langle object_write \rangle &\models (\langle write_region \rangle \langle object_write \rangle)* \\
\langle write_region \rangle &\models (\langle \text{id} \rangle \langle read_region \rangle \langle write_region \rangle)* \\
\langle read_region \rangle &\models \langle \text{id} \rangle \langle variable_name \rangle
\end{aligned}$$

Figure 4 illustrates the region node for the code example in Figure 1c. For this example, we have data writes to two primitive variables in the loop header from the collection variable `identifier_to_id` which is captured in the `primitive_write` subtree. In the body of the loop, there is a write to a collection variable `call_stack_nodes`. This dataflow is represented as a child node in the `collection_write` subtree. Since this is the first collection write in the loop body it is referenced with `collection_write_0` and the data reads from `identifier`, `id`, `nodes`, and `identifier_to_count` are represented as a right balanced subtree. The fix-point operation identifies that the variable write to `call_stack_nodes` also depends on data read from `identifier_to_id` since `identifier` and `id` data write depends on it. The canonicalization can be seen in the data read reference for `call_stack_nodes`, which is `collection_1` whereas the data write reference for the same variable is `collection_write_0`. Construction of similar canonicalized trees for other code snippets in Figure 1 helps identify the common pattern mentioned in Section 1. The pattern identified from these trees is that `foreach` region contains a `collection_write` to a `collection_0` variable with data read from `primitive_0`, `primitive_1` and `collection_0`.

Our representation captures information flow in addition to variable mutability, whereas CASTs [1] captures only variable mutability. Canonicalized labels map the first data write in Figure 1a, 1c, 1e to the same variable reference whereas CASTs maps them to different references (see Figure 2). *Jezero*'s tree representation arranges the dataflow information such that a *top-down* mining algorithm (Section 3.2) can efficiently extract frequent subtrees (i.e., frequent flow patterns) from large sets of trees.

3.2 Mining Idioms

Allamanis and Sutton [2] propose probabilistic tree substitution grammars to learn code idioms. A tree substitution grammar (TSG) is an extension to a context-free grammar (CFG), in which productions expand into tree fragments. Formally, a TSG is a tuple $G = (\Sigma, N, S, R)$, where Σ is a set of terminal symbols, N is a set of nonterminal symbols, $S \in N$ is the root of the nonterminal symbol and R is a set of productions. In case of TSG, each production $r \in R$ takes the form $X \rightarrow T_X$, where T_X is a tree fragment rooted at the nonterminal X. The way to produce a string from a TSG is to begin with a tree containing S, and recursively expand the trees — the difference is that some rules can increase the height of the tree by more than 1. A pTSG augments a TSG with probabilities, in an analogous way to a probabilistic CFG (pCFG). Each tree fragment in the pTSG can be thought of as describing a set of context-free rules that are used in a sequence. Formally, a pTSG is $G = (\Sigma, N, S, R, \sigma)$, which augments a TSG with σ, a set of distributions $P_{TSG}(T_X|X)$, for all $X \in N$, each of which is a distribution over the set of all rules $X \rightarrow T_X$ in R that have left-hand side X.

The goal of our mining problem is to infer a pTSG in which every tree fragment represents a code idiom. Given a set of trees $(T_1, ..., T_n)$ for pTSG learning, the key factor that determines model complexity is the number of fragment rules associated with each nonterminal. If the model assigns too few fragments to a non-terminal, it will not be able to identify useful patterns (*underfitting*); on the other hand, if it assigns too many fragments, then it can simply memorize the corpus (*overfitting*) [1]. Furthermore, we do not know in advance how many fragments are associated with each non-terminal. Non-parametric Bayesian statistics [9, 18] provide a simple, yet powerful, method to manage this trade-off for cases where the

Data Write	Data Read
(primitive_write_0, key)	(collection_0, identifiers)
(collection_write_0, values)	(collection_0,identifiers)
(primitive_write_1, exp)	(primitive_0,key)
(collection_write_1, results)	(primitive_0, key)

(a) Dataflow table of outer region before merge (σ_o)

Data Write	Data Read
(primitive_write_0, item)	(collection_0, values)
(collection_write_0, results)	(collection_0, values) (primitive_0, item), (primitive_1, exp)

(b) Dataflow table of inner (foreach) region (σ_i)

Data Write	Data Read
(primitive_write_0, key)	(collection_0, identifiers)
(collection_write_0, values)	(collection_0,identifiers)
(primitive_write_1, exp)	(primitive_0,key)
(collection_write_1, results)	(primitive_0, key) **(primitive_1, exp)**, **(collection_1, values)**

(c) Dataflow table of outer region after merge (σ_m)

Table 3: Dataflow tables of outer (σ_o), inner (σ_i) and merged (σ_m) regions for the code example in Listing 1

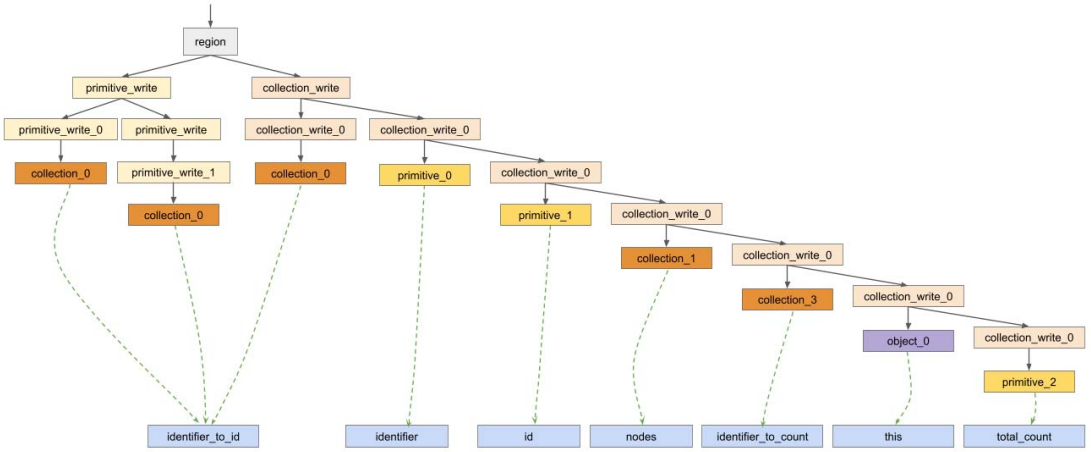


Figure 4: Region tree representing the dataflow operations for the code example in Figure 1c. Due to space limitations, we show a compact version of the AST instead; e.g., we collapse ids and rules into one node with the rule suffixed with id as label.

number of parameters is unknown. In this work, we use the non-parametric Bayesian inference methods proposed by Allamanis and Sutton [2] to mine refactoring idioms. To infer a pTSG G using Bayesian inference, we first compute a probability distribution over probabilistic grammars, $P(G)$. This distribution is bootstrapped by estimating the maximum likelihood from our training corpus. While this gives distribution over full trees, we require the distribution over fragments. This is defined as $P_0(T) = \prod_{r \in T} P(r)$, where r ranges over the set of productions that are used within T. The specific prior distribution that we use is Dirichlet process. The Dirichlet process is specified by a *base measure*, which is the fragment distribution P_0, and a *concentration parameter* $\alpha \in \mathbb{R}^+$ that controls the rich-get-richer effect. Given P_0 and prior distribution, we apply Bayes' rule to obtain posterior distribution. The posterior Dirichlet process pTSG is characterized by a finite set of tree fragments for each non-terminal. To compute this distribution, we resort to approximate inference based on Markov Chain Monte Carlo (MCMC) [13]. Specifically, we use Gibbs sampling to sample the posterior distribution over grammars.

At each sampling iteration, *Jezero* samples trees from the corpus and for each node in the tree it *decides* if it is a root or not based on posterior probability. *Jezero* adds trees to the sampling corpus and adds tree fragments to the sample grammar based on whether the fragments are root (denoted by $z_t = 1$) or not. Next, for each tree node T_t, *Jezero* identifies the parent T_s whose $z_s = 1$. Based on the current node and its root parent, *Jezero* samples it to merge them as a single fragment or to separate them into different fragments. To do this, *Jezero* computes the probability of the joint tree (node T_t and parent T_s), and the split probabilities. Based on a threshold it either splits into fragments or merges them into one fragment;

$$Pr(z_t = 0) = \frac{Pr_{post}(T_{join})}{Pr_{post}(T_{join}) + Pr_{post}(T_s) \cdot Pr_{post}(T_t)}$$

$$Pr_{post}(T) = \frac{count(T) + \alpha \cdot P_0(T)}{count(h(T)) + \alpha}$$

$T_{join} = T_t \cup T_s$, h is the root of the fragment, and count is the number of times that a tree occurs as a fragment in the corpus, as determined by the current values of z_t. Once the sampling is complete, *Jezero* orders the grammar based on the production probability and filter out those rules that have probability is less than 0.5.

In MCMC it is essential that there is good mixing of samples, hence *Jezero* visits the the trees in the corpus and their nodes in different orders to further introduce randomness. We seed the sampling process by annotating randomly 90% of the nodes with $z_t = 1$[3]. Furthermore, it incrementally adds trees to the corpus to compute the grammar. *Jezero* repeats this process for 50 iterations

[3]Other annotation values (namely, 40% and 60%) yield similar results at the cost of 3x slowdown in execution time.

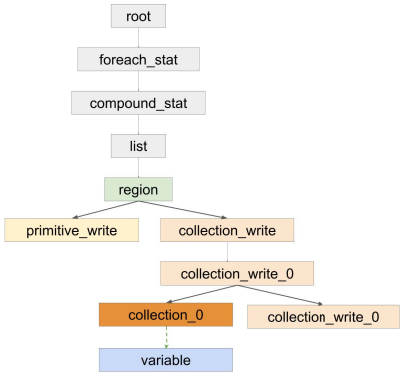


Figure 5: Example idiom mined by *Jezero*: Top-1 idiom for API Vec\map_map_with_key. An example of a code snippet for this idiom is in Figure 1a.

to identify the posterior distribution over fragments, which are then returned as idioms. We further experimented with 100 iterations but no changes in the evaluation scores for the APIs were observed.

Figure 5 shows the top-1 idiom mined by *Jezero* for the Vec\map_map_with_key API. **Note how this is prefix of the tree shown in Figure 4: the key advantage of the canonicalized representation.** Despite its shallow nature, in Section 4, we show that this idiom is very effective in identifying refactoring opportunities. In particular, according to this idiom, the most common dataflow pattern is a loop with a write to the first collection variable, and it depends on the loop iteration variable. The reason for *Jezero* to return shallow idioms is attributed to the Gibbs sampling process. Finding deeper idioms is possible by tuning the Gibbs sampling hyper-parameters (it remains, however, for future work).

4 EVALUATION

This section details the empirical evaluation of *Jezero* on the task of learning imperative idioms for a diverse set of APIs from the Facebook Hack codebase. These APIs are **Vec\map_with_key**, **Vec\gen_filter**, **Dict\filter** and **Dict\map_with_key**. Further, for each API, we measure the effectiveness of the mined idiom in identifying known and potential refactoring locations where imperative code can be replaced with the corresponding API call.

Dataset for Mining. As distinct APIs have different imperative code patterns, we need to construct a dataset of patterns, per API, using historical change data to prevent the sparsity problem(see 3.1) and to learn useful patterns. We call these changes *edits* and they contain a *before* and *after* version of changed source file(s). We first scrape edits in a given time interval and API from Facebook's code repository and construct a dataset for mining imperative code patterns using *before* versions of the edits. However, this suffers from a drawback that edits collected using this approach may contain excessive noise, i.e., changes that are not relevant to the imperative to API code changes. Hence, we opted for a relatively inexpensive way to prune out irrelevant edits. We propose the following three-fold heuristic filter, as shown in Figure 6: (1) first we compute a "treediff" of each edit using the GumTree algorithm [7] and remove method trees that were not modified or did not see an introduction of the API we are investigating, (2) we then collect code edits whose API keyword occurrence in after version is higher compared to before, (3) we further filter edits where the cyclomatic complexity of *before* is greater than the *after*. These heuristics are not meant to end up with the actual refactorings exclusively, but to increase the chances of each *before-after* pair being a valid refactoring. We believe that the resulting diversity in the dataset helps prevent overfitting.

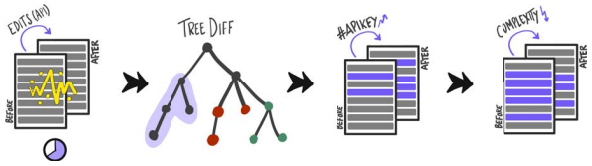


Figure 6: Phases of the mining dataset construction.

The initial dataset for the four APIs we used contains **21, 147** trees (average per API) rooted at the method level (i.e., number of methods—average per API—found before the three-fold heuristic). After the pruning stage, the dataset contains **1, 347** (average per API) trees rooted at method level (1, 347 trees for Vec\map_with_key; 1, 151 trees for Vec\gen_filter; 2, 198 trees for Dict\filter; 693 trees for Dict\map_with_key).

Experimental Setup. We evaluate the effectiveness of *Jezero* in two settings. First, we measure the accuracy of the proposed approach on a manually constructed validation set, containing true refactoring opportunities and non-opportunities. Second, we measure the performance of *Jezero* in identifying new refactoring opportunities in the entire codebase. For each API, we sample idioms for 50 iterations and with a concentration parameter value of 5.0. Further, we have pruned rare (`c_min` = 2) and small (`n_min` = 6) idioms. To surface interesting patterns, we use ranking schemes from [1, 2] and propose our own ranking scheme based on Jaccard similarity. F1 scores of *Jezero* and *Haggis* were identical across the different ranking schemes, and so we refrain from detailing them due to space limitations. The observations are obtained by running *Jezero* for 88 hours on an Intel Core Processor i7-6700 CPU @2.39GHz with 57GB RAM. Note that this is the *one-time* cost to train the four APIs. For each API, the training takes about the same amount (~22hours). Note that prediction time is just a few milliseconds to identify matching locations.

Effectiveness in identifying known refactoring. To measure accuracy, for each API, we manually construct a ground truth dataset (1) using manually confirmed refactoring locations in the historical change data and (2) manually identified potential refactoring locations from a set of files sampled from the current version of the codebase. These locations from the current codebase are included to get a wider variety of code samples.

The constructed evaluation dataset for this experiment (Table 4) contains **431** trees (average per API). This is the average number of trees randomly sampled from the 1, 347 trees mentioned before, as well as methods sampled from the current codebase. Of these, **27** (average per API) trees were manually verified to be true refactoring locations (see Figure 4). Note that not all 431 trees are true refactorings. This happens because, other than the true refactoring

method, an *edits*'s *before* version in our dataset may contain several methods with loopy code that is similar to the idiom. Hence, a manual check of the trees revealed that 27 were actual refactorings.

We compare *Jezero* with *Haggis*, an AST-based idiom mining proposed by Allamanis et.al [2]. *Haggis* does not have the dataflow augmentation, but is otherwise identical to *Jezero*. We could not compare with semantic idiom mining based on coiling [1] because the latter requires annotations or dynamic analysis. We measure the effectiveness of the proposed approach in identifying refactoring locations by comparing precision, recall, and F1 scores. Table 4 shows the accuracy results for four APIs when using top-1 idiom using the Jaccard similarity ranking scheme. On average *Jezero's* F1 score is significantly better than the baseline on all APIs. This shows the effectiveness of the proposed static analysis-based tree representation. The differences between the APIs also reflect how well the new tree representation can identify diverse patterns.

	Eval trees (true/total)	**Haggis** F1/precision/recall	***Jezero*** F1/precision/recall
Vec\map_with_key	36/437	0.14/0.43/0.08	0.71/0.74/0.69
Vec\gen_filter	22/535	0.0/0.0/0.0	0.50/0.39/0.72
Dict\filter	12/324	0.0/0.0/0.0	0.56/0.54/0.58
Dict\map_with_key	39/426	0.19/0.45/0.13	0.51/0.53/0.49
Average	27 / 431	0.08/0.22/0.05	0.57/0.55/0.62

Table 4: Performance of *Jezero* vs. Haggis.

Effectiveness in identifying new refactoring opportunities. In this experiment, we measure the performance of *Jezero* in identifying potential refactoring locations on Facebook's codebase with 13770 Hack methods, spread over 1501 files. For each API, we identify matching locations using the top-1 idiom from *Jezero* and *Haggis*. Table 5 summarizes the number of matching locations for each API. *Jezero* matches 807 locations (202 on average; i.e., 1.5% of the trees rooted at loop headers), whereas Haggis only matches 23 locations in all (6 on average); *Haggis* fails to identify *any* refactoring opportunities in the case of `Vec\gen_filter` and `Dict\filter`.

Further, to identify the precision of the matched locations, we manually inspect all locations of *Haggis* and a random subset of (100) locations for each API returned by *Jezero*. Note that since we

	#Matching Locations ***Jezero*** **/ Haggis**	**Precision** ***Jezero*** **/ Haggis**
Vec\map_with_key	260 / 1	0.91 / 1.00
Vec\gen_filter	247 / 0	0.41 / 0.00
Dict\filter	134 / 0	0.39 / 0.00
Dict\map_with_key	166 / 22	0.68 / 0.91
Average	202 / 6	0.60 / 0.48

Table 5: Top-1 idioms' matching locations in the wild.

do not have a dataset of locations that should match in the internal codebase, no measures of recall are reported. On average *Jezero* has a precision of 0.60, which is an encouraging number. Arithmetically, *Haggis*'s average precision works out to 0.48, but it is not meaningful to compare average precisions: for two of the APIs *Haggis* found only 23 refactoring opportunities compared to *Jezero* which found 426. Additionally, for two APIs *Haggis* found *zero* refactoring opportunities. **In summary, *Jezero* not only mined patterns in an unsupervised way, those mined patterns were extremely productive in locating opportunities in the "wild".**

Initial Feedback from Developers. We conducted an initial feasibility study to understand the usefulness of *Jezero* at Facebook. We reached out to 20 Facebook developers — each developer was shown one refactoring from the codebase authored by him/her — for *ad hoc* feedback on the potential refactoring locations (from Table 5) identified by *Jezero*. The locations identified by the tool were accepted by the developers as refactoring opportunities.

5 THREATS TO VALIDITY

Regarding internal validity, the effectiveness of parameters may depend on the extent and nature of codebase used. To mitigate this risk we have experimented with a combination of parameters and ranking schemes. However, we have not systematically explored every combination of parameters in our experiments. In terms of external validity, the proposed approach has only been evaluated using a codebase developed by a single company (albeit a large codebase). It has also been evaluated on a limited number of APIs, although we believe it should extend to similar APIs in a natural way. Also, the approach has been evaluated on the Hack programming language, and may only generalize to other programming languages with prudence. To mitigate this, as future work, we will investigate the effectiveness of our approach on other languages and codebases.

6 LIMITATIONS AND FUTURE WORK

The dataflow trees we generate are type agnostic. Therefore, different APIs could have similar idiomatic patterns. For example, we observe that top-3 idioms of `Vec\gen_filter`, `Dict\filter` are identical. To improve the precision of the proposed approach we can add type information while generating the tree representation, or use it to disambiguate APIs at prediction time. Moreover, the current dataflow trees are rather general — e.g., no information about if-expressions are captured. Adding more information like variable references in the `if` condition, will likely help mine better idioms. The proposed tree canonicalization was influenced by the idiom mining machinery which identifies contiguous patterns from trees. Capturing information about variables outside a local code block makes it a graph mining problem. To overcome this, we can introduce predicates, such as `contains,before,after`, and construct a tree based on this grammar. However, this might lead to a computationally expensive sampling approach.

The dataflow trees generated using a bottom-up approach where the information flow is in one direction, from the inner to the outer code block was a design choice we made to capture local patterns. This is not suitable for patterns that depend on context information from the outer region (see Section 3.1). Finally, refactoring opportunities identified by *Jezero* may not be good candidates for actual refactoring, due to the replacement API being less performant or readable. Therefore, we do not plan to automatically apply refactorings detected by *Jezero*, and instead surface suggestions to developers during the code review process.

7 RELATED WORK

Code clone detection [10, 12, 25] techniques are related to idiom mining, as the goal is to identify similar code blocks. Rattan *et al.* [20] identify several clone detection techniques that use syntax and semantics of a program [4, 11]. Code idiom mining proposed in this work searches for frequent as opposed to maximally identical subtrees as with clone detection techniques. Semantic code search techniques [6, 19, 26] are also related to idiom mining, since they utilize type [21], data, and control flow [19, 29] information for identifying clones. Our approach differs in two ways; (1) code search requires the user to provide a search pattern, whereas *Jezero* infers such a pattern (2) search techniques that infer a pattern [26] leverages active learning while we use nonparametric Bayesian methods. Another related area is API mining [14, 27, 30]. However, this problem is different from idiom mining since it tries to mine sequences or graphs of API method calls, usually ignoring most features of the language. API protocols can be considered a type of semantic idiom; therefore, idiom mining is a general technique for pattern matching and can be specialized to API mining by devising appropriate tree representations.

Recent years have seen an emerging trend of tools and techniques that synthesize program transformations from examples of code edits [3, 8, 17, 23]. The synthesized transformation should satisfy the given examples while producing correct edits on unseen inputs. Existing approaches have addressed this in different ways. Sydit [16] and LASE [17], are only able to generalize variables names, methods and fields. Moreover, the former only accepts one example and synthesizes transformations using the most general generalization, whereas the latter accepts multiple examples and synthesizes transformations using the most specific generalization. While these techniques learn transformations from the provided examples, *Jezero's* main focus is on the detection of statistically significant patterns from a corpus, and then pointing out likely opportunities for refactoring. On a different note, many of these tools can also benefit from the dataflow augmented tree structure that we introduced that makes the common semantic pattern manifest.

8 CONCLUSIONS

We propose *Jezero*, a scalable, lightweight technique that is capable of surfacing semantic idioms from large codebases. Under the hood, *Jezero* extends the abstract syntax tree with canonicalized dataflow trees and leverages a well-suited a nonparametric Bayesian method to mine the semantic idioms. Our experiments on Facebook's Hack code shows *Jezero*'s clear advantage. It was significantly more effective than the baseline that did not have the dataflow augmentation to find refactoring opportunities from unannotated legacy code. On a randomly drawn sample containing 13770 Hack methods, *Jezero* found matches at 1.5% locations, with a precision of 0.60 We expect the ideas in *Jezero* to carry over to other languages such as Python, as it provides ways to express idiomatic code.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. 2018. Mining semantic loop idioms. *IEEE Transactions on Software Engineering* 44, 7 (2018), 651–668.

[2] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 472–483.

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[4] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.

[5] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *The Journal of Machine Learning Research* 11 (2010).

[6] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[7] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*.

[8] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[9] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis*. CRC press.

[10] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 96–105.

[11] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering*. IEEE.

[12] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.. In *OSdi*, Vol. 4. 289–302.

[13] Percy Liang, Michael I Jordan, and Dan Klein. 2010. Type-based MCMC. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. 573–581.

[14] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. *ACM Sigplan Notices* 40, 6 (2005).

[15] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. 2015. Does automated refactoring obviate systematic editing?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 392–402.

[16] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices* 46, 6 (2011), 329–342.

[17] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511.

[18] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[19] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning.. In *PLDI*. 1066–1082.

[20] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013).

[21] Steven P Reiss. 2009. Semantics-based code search. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 243–253.

[22] Charles Rich and Richard C. Waters. 1988. The programmer's apprentice: A research overview. *Computer* 21, 11 (1988), 10–25.

[23] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[24] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

[25] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE*. IEEE, 1157–1168.

[26] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active inductive logic programming for code search. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 292–303.

[27] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Conference on Mining Software Repositories (MSR)*. IEEE, 319–328.

[28] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[29] 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 *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 457–466.

[30] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.