Mining Semantic Loop Idioms from Big Code

Miltiadis Allamanis *
University of Edinburgh, UK
m.allamanis@ed.ac.uk

Earl T. Barr
University College London, UK
e.barr@ucl.ac.uk

Christian Bird
Microsoft Research, USA
cbird@microsoft.com

Premkumar Devanbu
UC Davis, USA
devanbu@ucdavis.edu

Mark Marron
Microsoft Research, USA
marron@microsoft.com

Charles Sutton
University of Edinburgh, UK
csutton@ed.ac.uk

Abstract

During maintenance, developers spend a lot of time transforming existing code: refactoring, optimizing, and adding checks to make it more robust. Much of this work is the drudgery of identifying and replacing specific patterns, yet it resists automation, because of meaningful patterns are hard to automatically find. We present a technique for mining loop idioms, surprisingly probable semantic patterns that occur in loops, from big code to find meaningful patterns. First, we show that automatically identifiable patterns exist, in great numbers, with a large scale empirical study of loop over 25 MLOC. We find that loops in this corpus are simple and predictable: 90% of them have fewer than 15LOC and 90% have no nesting and very simple control structure. Encouraged by this result, we *coil* loops to abstract away syntactic diversity to define information rich loop idioms. We show that only 50 loop idioms cover 50% of the concrete loops. We show how loop idioms can help a tool developers identify and prioritize refactorings. We also show how our framework opens the door to data-driven tool and language design discovering opportunities to introduce new API calls and language constructs: loop idioms show that LINQ would benefit from an `Enumerate` operator, a result confirmed by the fact that precisely this feature is one of the most requested features on StackOverflow with 197 votes and 95k views.

1. Introduction

When coding, developers often rewrite and transform existing code to optimize it, increase its API conformance, or refactor it. These tantalize with mechanical steps, yet resist automation because the patterns to rewrite are so numerous and varied. Compiler optimization illustrates the difficulty of selecting and covering high-yield patterns. Despite 40 years of research, the cost of handling them has prevented compilers from performing all the optimizations that are within the reach of automated analysis. A study of vectorizing compilers, which rewrite sequential loops to use vector instructions, found that, while collectively the compilers successfully rewrote 83% of the benchmark loops, each individual compiler vectorized only 45–71% (Maleki et al. 2011) of those loops.

Designers and tool developers who work to manipulate code seek *useful patterns*: patterns that are easy to manipulate and reason about; patterns that match syntactic terms that implement a cohesive functionality. Tools such as `grep` and manual inspection currently dominate the search for useful patterns and, unfortunately, tend to return frequent but trivial or redundant patterns. Tool developers need help finding useful, meaningful patterns. This work seeks to find these patterns.

To this end, we propose *semantic idiom mining*, a new technique for automatically mining meaningful patterns, so designers and tool developers can prioritize them. Semantic idioms are patterns that abstract the concrete execution traces of the loops they match (Cousot and Cousot 1977) to provide meaningful high-level representations. This representation is designed to encapsulate properties relevant to a tool developer's needs, eliminating other unnecessary information. Humans aggregate concepts and data into mental chunks (Guida et al. 2013). Consider a compiler developer who has written a loop to algebraically simplify an instruction sequence. When talking to another developer, the developer might describe the loop as "algebraically simplifying arithmetic instructions". Ideally, our idioms capture these mental chunks over code.

Previous work mined *frequent* patterns to find *meaningful* patterns. Unfortunately, the "a priori pinciple" means that the larger a pattern, the smaller its support, *i.e.* the number of objects covered by that pattern

* Work done primarily while author was an intern at Microsoft Research, WA, USA.

```
foreach (var 0 in EXPR)
   $REGION[UR(0, 1); URW(2);]
```

(a) A semantic loop idiom capturing a reduce idiom, which reads the unitary variables 0 amd 1 and reduces them by writing into the unitary variable 2.

```
foreach(var refMap0 in mapping.ReferenceMaps)
    this2.AddProperties(properties1,
                        refMap0.Data.Mapping);
```

(b) Concrete loop from csvhelper that match the semantic loop idiom in Figure 1a.

Figure 1: A semantic idiom and a matching loop. For more samples, see Figure 4.

(Aggarwal and Han 2014, Chap. 5). Further, frequent pattern mining does not capture statistical dependence among the mined elements. Thus, putting a floor under pattern size triggers *pattern explosion*, returning an unwieldy number of patterns that differ only trivially (Fowkes and Sutton 2016). For a concrete example from our corpus, consider `foreach(var v in c){}` which has an unspecified loop body. It is a very frequent, but meaningless, pattern. In short, frequent patterns are rarely meaningful to developers.

Allamanis and Sutton (2014) sought to find meaningful patterns in big code by mining syntactic idioms, code patterns that do capture usage dependencies among mined elements. Although syntactic idioms are more likely to be useful than frequent patterns, data sparsity, exacerbated by the commendable practice of code reuse (e.g. sorting algorithms), means that many syntactic idioms often fall short of being meaningful, as you can see in this example, from Allamanis and Sutton (2014):

```
FileSystem name= FileSystem.get([Path].toUri(), conf);
```

where `name` and `[Path]` are meta-variables. Specifically, few syntactic idioms meaningfully contained loops at all, let alone a loop that performs a reduce operation.

In this work, we combat this sparsity with abstraction. Our abstract representation removes information irrelevant to tool developers. This step is crucial for creating a representation that is semantically and syntactically meaningful and contains a rich structure well-suited to machine learning methods. We focused our mining on coiled loops (subsection 3.2), because loops are vital to programming and program analysis. Our coiling abstraction removes syntactic information, such as variable and method names, while retaining loop-relevant properties like loop control variables, collections, and purity information. Loop idiom mining finds meaningful patterns, such as the simple reduce idiom in Figure 1 and a concrete loop it matches. Further sample idioms and loops can be found in Figure 4.

To approximate semantic properties like purity during the coiling process, we used testing because its speed in practice allows it to scale to "Big Code" and, crucially, because one can easily test industrial code. Concerning its speed, Beller et al. (2016) found that 75% of the top projects in GitHub require less than 10 minutes to execute the full test suite, many of which comprise more than 500 tests. Essentially, we check whether a property holds modulo a test suite. Although this process is unsound, it is effective in our setting where we pipe its output to machine learning, which robustly handles "noise" (*e.g.* misclassifications) in data.

To show that loop idioms exist in sufficient numbers to make mining them useful, we conducted a large-scale empirical study of loops across a corpus of 25.4 million LOC containing about 277k loops (Section 4). Our key finding is that real-life loops are mostly simple and repetitive. For example, 90% of loops have no nesting, are less than 15 LOC long and contain very simple control-structure.

Despite their regularity, loops also have a heavy tail of diversity, exhibiting nontrivial variability across domains: on average, 5% and, for some projects, 18% of loops are domain-specific. For example, we find that loops in testing code are much shorter than loops in serialization code, while math-related loops exhibit more nesting than loops that appear in error handling (Table 2). Loop idioms capture sufficient detail to identify useful-patterns despite this diversity. They identify opportunities to replace loops with functional operators, while remaining general enough to cover most loops: 100 idioms capture 62% and 200 idioms capture 70% of all loops in our corpus. To demonstrate the utility of mining and ranking semantic loop idioms, we present three case studies that exploit loop idioms to suggest refactorings, new language constructs, or APIs.

First, we built and evaluated an engine (subsection 6.1) that suggests replacing a loop with a functional construct in LINQ[1]. This refactoring suggestion engine matches a concrete loop to a loop idiom, then looks up that idiom in a manually constructed table of equivalent LINQ statements. Its suggestions are context-insensitive and unsound; it is aimed at a refactoring tool developer, not developers directly. A refactoring toop developer could use it to identify which loop patterns to target. Over our corpus, we manually mapped the top 25 idioms to LINQ statements and covered 45.4% of all the concrete loops in 12 hours. The resulting suggestion engine correctly suggested LINQ replacements for loops 89% of the time as judged by human annotators. Loop idioms could similarly help vectorizing compilers, where they

[1] Language Integrated Query (LINQ) is a .NET extension that provides functional-style operations, such as map-reduce, on streams of elements and is widely used in C# code.

could identify and rank loop patterns whose support would most improve a compiler's coverage of loops.

Second, mining semantic idioms identifies opportunities for new API features that can significantly simplify existing code (subsection 6.2). For example, we found that in lucenenet adding an API method that accepts a collection of elements would simplify API usage that currently requires adding single elements. We also make similar observations within mathnet-numerics, where new APIs could vastly simplify code.

Finally, semantic idioms can also be useful for guiding programming language design (subsection 6.3). Java's `foreach` and multicatch constructs simplify common idioms that our framework could have identified automatically. Their successful adoption illustrates how loop idioms could help language designers. Had our framework been available, designers could have used it to spot the need for these constructs, speeding their implementation and deployment. Our idiom mining has identified such opportunities in C# and LINQ. A common operation is tracking the index of each element in a collection during traversal. Adding an `Enumerate` operator to C#, similar to Python's, would allow developers to perform this operation more succinctly: adding `Enumerate` to C# would simplify 12% of loops in our 25.4 MLOC corpus.

This paper presents a principled and data-driven approach to support the construction of code transformation and analysis tools. Increasing the productivity of tool developers promises to bring domain-specific, even project-specific, tools within reach at reasonable cost; it also is a first step toward data-driven language and API design. Our principal contributions follow:

- We introduce semantic idiom mining, a new technique for mining big code for semantic idioms, and specialize it for loop idioms, a code abstraction that captures semantic loop properties (Section 3);
- We conduct a large-scale study of 277k loops in a corpus of 25.4 MLOC showing that most loops are surprisingly simple with 90% of them having less than 15 LOC and no nesting (Section 4); and
- We demonstrate the utility of loop idioms for tool and language construction via three case studies: two centered on language and API design, showing that adding `Enumerate` to C# would simply 12% of loops, and the other on refactoring, showing that the 25 loop idioms cover 45% of concrete loops and suggest loop-to-LINQ refactorings with 89% accuracy (Section 6).

Our data and code is available online at `http://groups.inf.ed.ac.uk/cup/semantic-idioms/`.

2. Background

We are interested in the problem of extracting meaningful idioms from a code base. In machine learning, this is an unsupervised learning problem. The semantic idiom mining method we use in this paper builds on the work of Allamanis and Sutton (2014). Idiom mining is a way of looking for patterns that can compactly encode the training set as the concatenation of those patterns with their composition. Consider the minimum length description (MLD) of a program; the length of this description is the program's Kolmogorov complexity. Each symbol in the MLD encodes a pattern in the program. To achieve minimality, these symbols must collectively balance frequency and support. Idiom mining is analogous to finding symbols in the MLD of a program. Thus, idioms are surprisingly probable in their context, but not necessarily frequent, patterns. In contrast, frequent pattern mining tends to miss many patterns that are present in the data (Kuzborskij 2011). We employ probabilistic tree substitution grammar (pTSG) inference that automatically and exhaustively captures the idioms needed to reconstruct a forest of ASTs.

Why These Methods? It is certainly worth asking why powerful statistical methods are necessary for the idiom mining problem, rather than simpler methods that are easier to apply. First, one might ask: why employ a *probabilistic* model here? The reason is that probabilities provide a natural quantitative measure of the quality of a proposed idiom. Imagine that we create two different models, one M_1 that contains a proposed idiom and another M_2 without it. Then for each of these alternative models, we can measure which has a higher value under the posterior distribution, and this measures the value of the proposed idiom. In other words, a proposed idiom is worthwhile only if, when we include it into a pTSG, it increases the probability that the pTSG assigns to the training corpus. This encourages the method to avoid identifying idioms that are frequent but trivial and meaningless. As we show below, the statistical procedure that we in fact employ is quite a bit more involved, but this is a good basic intuition. Second, it may seem odd that we are applying grammar learning methods when the grammar of the programming language is already known. However, our aim is *not* to re-learn the known grammar, but rather to learn probability distributions over ASTs from a known grammar. These distributions represent which rules from the grammar are used more often and, crucially, which sets of rules tend to be used contiguously.

2.1 Representing Idioms

We represent idioms as fragments of a tree structure (*e.g.* an AST). More precisely, a fragment is a connected subgraph of the valid syntax tree of some string

in the language. It is easy to see how the simple `for(int=0;i<n;i++){BODY}` loop can be represented by a fragment whose root is of type `ForLoop`, say. One nice aspect of the fragment representation is that leaf nodes in a fragment that correspond to *non*terminals in the language grammar are exactly the metavariables of the idiom. For example, the body portion of the for-loop example can be represented as a leaf node in the fragment whose type is `BlockStatement`. In particular, we describe a probabilistic tree substitution grammar (pTSG), which is a probabilistic context free grammar with the addition of special rules that insert a tree fragment all at once.

To fix notation, a *probabilistic context free grammar* (PCFG) defines a distribution over the strings of a context-free language. A PCFG is defined as $G = (\Sigma, N, S, R, \Pi)$, where Σ is a set of terminal symbols, N a set of nonterminals, $S \in N$ is the root nonterminal symbol and R is a set of productions. Each production in R has the form $X \to Y$, where $X \in N$ and $Y \in (\Sigma \cup N)^*$. The set Π is a set of distributions $P(r|c)$, where $c \in N$ is a nonterminal, and $r \in R$ is a rule with c on its left-hand side. To sample a tree from a PCFG, we recursively expand the tree, beginning at S. Each time we add a nonterminal c to the tree, we expand c using a production r that is sampled from the corresponding distribution $P(r|c)$. The probability of generating a particular tree T from this procedure is the product over all rules that are required to generate T.

A tree substitution grammar (TSG) (Joshi and Schabes 1997; Cohn et al. 2010; Post and Gildea 2009) is a simple extension to a CFG, in which productions expand into tree fragments rather than simply into a list of symbols. Formally, a TSG is also a tuple $G = (\Sigma, N, S, R)$, where Σ, N, S are exactly as in a CFG, but now each production $r \in R$ takes the form $X \to \mathcal{T}_X$, where $\mathcal{T}_X$ is a tree fragment rooted at the nonterminal X. To produce a string from a TSG, we begin with a tree containing only S, and recursively expand the tree top-to-bottom, left-to-right as in CFGs — the only difference is that some rules can increase the height of the tree by more than 1. A probabilistic tree substitution grammar (pTSG) G (Cohn et al. 2010; Post and Gildea 2009) augments a TSG with probabilities, in an analogous way to a PCFG. A pTSG is defined as $G = (\Sigma, N, S, R, \Pi)$ where Σ is a set of terminal symbols, N a set of non terminal symbols, $S \in N$ is the root nonterminal symbol, R is a set of tree fragment productions. Finally, Π is a set of distributions $P_{TSG}(\mathcal{T}_X|X)$, for all $X \in N$, each of which is a distribution over the set of all rules $X \to \mathcal{T}_X$ in R that have left-hand side X.

The key reason that we use pTSGs for idiom mining is that each tree fragment $\mathcal{T}_X$ can be thought of as describing a set of context-free rules that are typically used in sequence. This is exactly what we are trying to discover in the idiom mining problem. In other words, *our goal is to induce a pTSG in which every tree fragment represents a code idiom* if the fragment has depth greater than 1 — we call these rules *fragment rules*. The remaining TSG rules, those whose RHS has depth 1, are less interesting, as they are simply the productions from the original CFG of the programming language. As a simple example, consider the PCFG

$E \to E + E$	(prob 0.7)	$T \to F * F$	(prob 0.6)
$E \to T$	(prob 0.3)	$T \to F$	(prob 0.4)
$F \to (E)$	(prob 0.1)	$F \to id$	(prob 0.9),

where E, T, and F are nonterminals, and E the start symbol. Now, suppose that we are presented with a corpus of strings from this language that include many instances of expressions like $id * (id + id)$ and $id * (id + (id + id))$ (perhaps generated by a group of students who are practicing the distributive law). Then, we might choose to add a single pTSG rule to this grammar, like

$E \to F * (T + T)$	(prob 0.4)		
$E \to E + E$	(prob 0.3)	$E \to T$	(prob 0.3)

When we add the pTSG rule, we adjust the probabilities of the previous rules so that all of E's productions sum to 1 as shown. Essentially, this allows us to a represent a correlation between the rules $E \to T + T$ and $T \to F * F$. Finally, note that every CFG can be written as a TSG where all productions expand to trees of depth 1. Conversely, every TSG can be converted into an equivalent CFG by adding extra nonterminals (one for each TSG rule $X \to \mathcal{T}_X$). So TSGs are, in some sense, fancy notation for CFGs. This notation will prove very useful, however, when we describe the learning problem next.

2.2 Inferring Idioms

To solve the idiom mining problem, a natural idea is to search for subtrees that occur often in a corpus. However, this naïve method does not work well, for the simple reason that frequent patterns can be meaningless patterns. This is a well-known problem in data mining (Aggarwal and Han 2014, Chap. 5). To return to our previous example, `for` loops occur more commonly than "`for(int i=0;i<n;i++) {BODY}`", but it would be hard to argue that "`for(INIT; COND; UPD) {BODY}`" on its own (with no expressions or body) is an interesting pattern. Instead, Allamanis and Sutton (2014) suggest a different principle: interesting patterns are those that help to explain the code that programmers write. It is when it comes to quantifying the phrase "help to explain" that the machinery of statistical natural language processing becomes necessary. Essentially the goal is that each returned idiom correspond to a group of syntactic rules

that often co-occur. To formalize this intuition, the idea is to infer a pTSG that is equivalent to the original language grammar in the sense of generating the same set of strings, but provides a better explanation of the data in the statistical sense. We do this by learning a pTSG that best explains a large quantity of existing source code. We consider as idioms the tree fragments that appear in the learned pTSG. We learn the pTSG using a powerful framework called *nonparametric Bayesian methods*.

Nonparametric Bayesian methods provide a theoretical framework to infer how complex a model should be from data. Adding parameters (which correspond to pTSG fragment rules in our case) to a machine learning model increases the risk of overfitting the training data, simply by memorizing it. But if we allow too few parameters, then the model will be unable to find useful patterns (*i.e.* underfit). Bayesian statistics (Gelman et al. 2013; Murphy 2012) provides a simple and powerful method to manage this trade-off. The basic idea is that whenever we want to estimate an unknown parameter θ from a data set $x_1, x_2, \ldots x_N$, we should not only treat the data as random variables — as in classical statistics — but also θ as well. To do this, we must choose a prior distribution $P(\theta)$ encoding any prior knowledge about θ, and then a likelihood $P(x_1 \ldots x_N \,|\, \theta)$ that describes a model of how the data can be generated given θ. Once we define a prior and a likelihood, we can infer θ via its conditional distribution $P(\theta | x_1 \ldots x_N)$ by Bayes' rule. This distribution is called the *posterior distribution* and encapsulates all of the information that we have about θ from the data. We can compute summaries of the posterior to make inferences about θ. For example, if we want to estimate θ by a single vector, we might compute the mean of $P(\theta | x_1 \ldots x_N)$. To summarize, applications of Bayesian statistics have three steps: 1) choose a prior $p(\theta)$; 2) choose a likelihood $p(x_1 \ldots x_N \,|\, \theta)$; and 3) compute $p(\theta | x_1 \ldots x_N)$ using Bayes's rule.

As a simple example, suppose the data $x_1 ... x_N$ are real numbers, which we believe to be distributed independently according a Gaussian distribution with variance 1 but unknown mean θ. Then we might choose a prior $p(\theta)$ to be Gaussian with mean 0 and a large variance, to represent the fact that we do not know much about θ before we see the data. Our beliefs about the data indicate that $p(x_i | \theta)$ is Gaussian with mean θ and variance 1. By applying Bayes's rule, it is easy to show that $P(\theta | x_1 \ldots x_N)$ is also Gaussian, whose mean is approximately[2] equal to $N^{-1} \sum_i x_i$ and whose variance is approximately $1/N$. This distribution represents a Bayesian's belief about the unknown mean θ, after seeing the data.

[2] The exact value depends on precisely what variance we choose in $p(\theta)$, but the formula is simple.

Nonparametric Bayesian methods handle the more complex case where the *number* of parameters is unknown as well. They focus on developing prior distributions over infinite dimensional objects (*e.g.* the infinite set of possible pTSG rules in our case), which are then used within Bayesian statistical inference. Bayesian nonparametrics have been the subject of intense research in statistics and machine learning (Gershman and Blei 2012; Teh and Jordan 2010). To infer a pTSG G using Bayesian inference, our prior distribution must be *a probability distribution over probabilistic grammars*, which we call $P(G)$. A sample from $P(G)$ is a pTSG, which is specified by the set of fragments $\mathcal{F}_X$ that are rooted at each nonterminal X, and a distribution $P_{TSG}(\mathcal{T}_X | X)$ over the rules that can be used to expand each nonterminal X. Sampling this pTSG gives us full trees. The specific prior distribution that we use is called a *Pitman-Yor process*. This choice was based on previous work in applying pTSGs to natural language (Post and Gildea 2009; Cohn et al. 2010). Briefly, the Pitman-Yor process prior has the following properties 1) There is no a priori upper bound on the size of the pTSG (that is, the method is *nonparametric*). 2) It favors grammars that are not too large, creating a penalty that discourages the method from memorizing the training set. 3) It allows to model *Zipfian* distribution of productions, that is, that the top few productions are used very frequently, while the great majority of idioms are used less commonly. This seems natural since it is well known that both source code and natural language display Zipfian properties in their token distributions. Here, we differ from Allamanis and Sutton (2014) by using the more general Pitman-Yor process (instead of its simpler Dirichlet process) and the fact that we do not assume a geometric distribution over the number of productions in the prior.

Given the prior distribution over pTSGs $P(G)$, we can apply Bayes's rule to obtain a posterior distribution $P(G | T_1, T_2, \ldots T_N)$. Intuitively, this distribution represents for every possible pTSG G, how much we should believe that G generated the observed data set. Applying Bayes's rule, the posterior is

$$P(G | T_1, T_2, \ldots T_N) = \frac{\prod_{i=1}^{N} P(T_i | G) p(G)}{p(T_1, T_2, \ldots T_N)}$$

i.e. it assigns high probability to grammars G that themselves assign high probability to the data (this is $P(T_i | G)$) and that receive a high score according the prior distribution $p(G)$. Unfortunately, the posterior distribution cannot be efficiently computed exactly, so — as is common in machine learning — we resort to approximations. The most commonly used approximations in the literature are based on Markov chain Monte Carlo (MCMC). MCMC is a family of randomized method that runs for a user-specified number of iterations. At

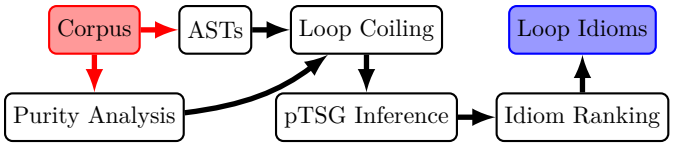


Figure 2: The architecture of our semantic idiom mining system, specialized to loop idioms. As Section 5 demonstrates, loop idioms enable code transformation tool developers and language designers to make data-driven decisions about which rewritings or constructs to add.

each iteration t, MCMC generates a pTSG G_t that has the special property that if t is taken to be large enough, eventually the sampled value G_t will be approximately distributed as $P(G|T_1,T_2,\ldots T_N)$. In our work, we use an MCMC method (Liang et al. 2010) for as many iterations t as we can afford computationally and then extract idioms from the few final samples of the pTSG.

3. Methodology

In this work, we mine loop idioms, semantic idioms restricted to loop constructs. We focus on loops because they represent an ubiquitous code construct that resists analysis, but the process can be easily generalized to other code constructs. Figure 2 depicts the workflow of loop idiom mining. Below, we discuss how the mined loop idioms look like and how refactoring, language, and API can use them to identifying candidate refactorings, language constructs, or method calls. To mine loop idioms, we first manipulate the ASTs into coiled ASTs (CAST), AST-like structures that preserve and add semantic information (*e.g.* variable purity information) and remove variability irrelevant to the loop semantics (*e.g.* names and exact operations are removed). The CASTs are then passed to the pTSG inference to mine the idioms.

3.1 Purity Analysis

Purity information is essential for loop semantics and we embed this information in the CASTs. For us, a function is *pure* in a variable (or global), when it is does not write to that variable during its execution. *Impurity*, its complement, is a strong property of code. Usually only a few runs of a code fragment are necessary to reveal impurity, because impure code must be carefully written to disguise its impurity and there is rarely any reason to do so. Thus, exercising a code snippet against its program's test suite is likely detect its impurity. Armed with this intuition, we implement an approximate dynamic purity detection technique. Given a method and a test suite that invokes that method, we run the test suite and snapshot memory before and after each invocation of the method. If the memory is unchanged across all its invocations, the method is *pure modulo the test suite*; otherwise, it is impure.

To snapshot the heap, we traverse the heap starting at the input method's reference arguments and globals. The heap is an arbitrary graph, but we traverse it breadth first without looping at backedges and compute its hash. We compare the hashes of the before and after invocation snapshots, to infer variable granular purity. If the test suite does *not* execute the method, its purity, and that of its variables and globals, is unknown. Otherwise, the input method's arguments and globals are pure until marked impure. When it executes, our technique may report false positives (incorrectly reporting a variable as pure, when it is impure) but not false negatives.

Our use of a dynamic purity analysis avoids the imprecision issues common to static analyses (Xu et al. 2007) and is sufficient for mining semantic loop idioms within big code since machine learning is robust to small amounts of noise. Other applications may require soundness; for this reason, we designed our mining procedure to encapsulate our dynamic purity analysis so that we can easily replace it with any sound static analysis (Sălcianu and Rinard 2005; Marron et al. 2008; Cherem and Rugina 2007), without otherwise affecting our idiom mining. An important aspect of inferring the useful idioms is to annotate their syntactic structure with semantics information, as we do here with purity information. It would be easy to further augment idioms with other semantically rich properties, such as heap/aliasing information (Barr et al. 2013; Raychev et al. 2014).

We instrument every method to realize our technique. First, we wrap its body in a `try` block, so that we can capture all the ways the function might exit in a `finally` block. At entry and in the `finally` block, we then inject snapshot calls that take the method's arguments and globals and computes their hash. In the `finally` block after the snapshot, we compare the hashes and mark any variables that point to memory that changed as impure. Once a variable is marked impure, no further instrumentation is made on that variable.

To speed up our purity inference and avoid the costly memory traversals, we use exponential backoff: if a method has been tested n times and has been found pure with respect to some argument or global, then we test purity only with probability p^n. We used $p = 0.9$. As a further optimization and to avoid stack overflows, we assume that by convention the overriden methods `GetHashCode()` and `Equals(object)` methods to be pure and ignore them. These methods may be invoked very frequently and therefore instrumentation is costly. Our method cannot detect when a variable is overwritten with the same value. This is a potential source of false positives to the extent to which such identity rewritings are correlated with impurity.

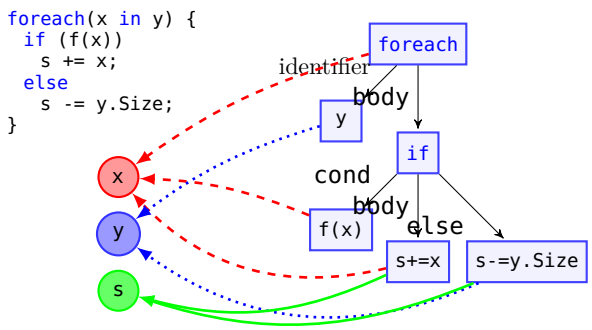


Figure 3: Abstract Syntax Tree with References.

Since we cannot easily rewrite and rebuild libraries, our technique cannot assess the purity of calls into them. However, they are frequent in code, so we manually annotated the purity of about 1,200 methods and interfaces in core libraries, including CoreCLR. These annotations encompass most operations on common data structures such as dictionaries, sets, lists, strings *etc.*

3.2 Coiling Loops

In this work we are interested in mining semantic loop idioms to capture universal semantic properties of loops. Thus, we keep only AST subtrees rooted at loop headers and abstract nodes that obscure structural similarities. We call this process *coiling* and detail its abstractions next. Coiling is a series of transformations on the AST and its output is another tree structure, which is the input to the idiom miner (as described in Section 2).

Program Variables In conventional ASTs, a node refers to a single variable. Coiling breaks this rule, creating nodes that potentially refer to many variables. Because we want to infer loop idioms that incorporate and therefore match patterns of variable usage, we need to re-encode variable usage into the AST. To this end, we introduce the notion of a reference. A *reference* is a set of nodes that refer to the same program variable. We label nodes with zero or more references as depicted in Figure 3. To combat sparsity, our pTSG inference merges two references that share the same node set. Thus, an idiom can match a concrete loop that contains more variables than the number of references in the idiom.

Expressions Expressions are quite diverse in their concrete syntax, due to the diversity of variable names, but are usually structurally quite similar. Since our goal is to discover universal loop properties, we abstract loop expressions into a single `EXPR` node, labeled with the variables that it uses. There are three exceptions: increment, decrement, and loop termination expressions. The pre and post increment and decrement operators from C introduce spurious diversity in increment expressions. Thus, we abstract all increment and decrement operations to the single `INC/DEC` node. We preserve the top-level operator of a termination expression and rewrite its operands to `Expr` nodes, with the exception of the common bounding expressions that compute a size or length, which we rewrite to a `SizeOf` node and label it with the reference to the measured variable.

Regions A region is a sequence of lines of code lacking control statements. Regions tend to be quite diverse, so we collapse region AST subtrees into a single node labeled with references to the variables they use. These regions are equivalent to uninterpreted functions. To make our pTSG inference aware of purity, we encode the purity of each of a region's variables as children of the region's node. We label each child node with its variable's reference and give it a node type that indicates its purity in the region. The purity node types are R, W, and $\boxed{\text{RW}}$.

Loops usually traverse collections, so we distinguish them from unitary (primitive or non-collection) variables. U denotes a unitary variable. For collection variables (denoted by C), we separate them into their *spine*, the references that interconnect the elements, and the elements it contains. Our purity analysis separately tracks the mutability of a collection's spine C^S and its elements C^E. This notation allows us to detect that a collection has changed when the same number of elements have been added and removed, without comparing its elements to the elements in a snapshot. In practice, the spine and the elements change together most often and only 9 idioms of the top 200 idioms (with total coverage 1.2%) have loops that change the elements of a collection, but leave the spine intact.

Blocks Blocks — the code that appears between `{` and `}` delimiters in a C-style language — can have multiple exits, including those, like `break` or `continue` statements, that exit the loop. Coiling transforms block nodes into two types of nodes: single and multi-exit blocks. This allows our pTSG inference to infer loop idioms that distinguish single exit blocks, whose enclosing loops are often easier to refactor or replace.

3.3 Idiom Ranking

We mine the idioms from CASTs as described in Section 2. After mining the idioms, we rank them in order of their utility in characterizing the target constructs — loops in our case. The ranked list provides data-based evidence to interested parties (*e.g.* API designers, refactoring tool developers) augmenting their intuition when identifying the most important code constructs. To mine idioms, we use a score that computes a trade-off between coverage and idiom expressivity. If we solely ranked idioms by coverage, we would end up picking very general and uninformative loop idioms, as would happen with frequent tree mining. We want idioms that have as much information content as possible *and* the greatest possible

	Semantic Idiom	Sample Matching Concrete Loop	Semantic Operation	Coverage
(1)	`for(int 0=EXPR; 0<EXPR; INC(0)) $REGION[UR(0);C^{S,E}R(1); URW(2)]`	`for (int i0 = 0; i0 < length; i0++) charsNeeded2 += components1[i0].Length;`	Reduce with `for`	14%
(2)	`foreach(var 0 in EXPR) $REGION[UR(0, 1); URW(2);]`	`foreach(Term term0 in pq.GetTerms()) rootMap2.AddTerm(term0.Text, query1.Boost);`	Reduce with `foreach`	2%
(3)	`foreach(var 0 in EXPR) $REGION[UR(0, 1);C^{S,E}RW(2)]`	`foreach(DictionaryEntry entry0 in dict) hash2[entry0.Key]=entry0.Value;`	Map with `foreach`	2%
(4)	`foreach(var 0 in EXPR) $REGION[UR(1); URW(0, 2);]`	`foreach(var exp0 in args) exp0.Emit(member1, gen2);`	Map overwrite and reduce with `foreach`	2%
(5)	`for(int 0=EXPR; 0<EXPR; INC(0)) $REGION[UR(0,1);C^{S,E}R(2);C^{S,E}RW(3)]`	`for (int k0=a; k0<b; k0++) ranks3[index2[k0]] = rank1;`	Map collection-to-collection with `for`.	5%
(6)	`for(int 0=EXPR; 0<EXPR; INC(0)) $REGION[UR(0,1);URW(2);C^{S,E}RW(3)]`	`for (var k0= 0; k0<i; k0++){ d3[k0] /= scale1; h2 += d3[k0] * d3[k0]; }`	Map and reduce with `for`.	5%
(7)	`foreach(var 0 in EXPR) $REGION[UR(1);URW(0)]`	`foreach(LoggingEvent event0 in loggingEvents) event0.Fix = m_fixFlags1;`	Map and overwrite `foreach`.	1%
(8)	`for(var 0=0; 0 < $EXPR(1,2,3); INC(0)){ if($EXPR(0, 1, 2, 4, 5)) $REGION[UR(0, 1);URW(4);C^{S,E}R(2)] }`	`for(int i0=0; i0<data2.Length3; i0++){ if(data2[i0]>max4 && !float5.IsNaN(data2[i0])) max4 = data2[i0]; }`	Reduce with `for` and conditional	1%

Figure 4: Semantic idioms automatically mined by our method and ordered using our ranking method. For each idiom we include a sample concrete loop it matches. Some concrete loops were slightly modified to fit the table and reduce their size (removed braces, shortened variable names). Idiom metavariables are highlighted with a colored box and a unique reference number is assigned to them. The same numbers appear within the concrete loops next to each variable, indicating each variable's binding to a metavariable. Non-terminals (*e.g.* EXPR) are also denoted within the colored box. Idiom (2) is the one shown in Figure 1. The `this` unitary variable is implied in some contexts.

coverage. We score each idiom by multiplying the idiom's coverage with its cross-entropy gain. Cross-entropy gain measures the expressivity of an idiom and is the average (over the number of CFG productions) log-ratio of the posterior pTSG probability of the idiom over its PCFG probability. This ratio measures how much the idiom improves upon the base PCFG. To pick the top idiom we use the following simple iterative procedure. First, we rank all idioms by their score and pick the top idiom. Then, we remove all loops that were covered by that idiom and repeat the process. We repeat this until there are no more loops covered by the remaining idioms. This greedy knapsack-like selection, yields idioms that achieve both high coverage and are highly informative. Since purity information is explicitly encoded within the CASTs (as special nodes, as discussed in subsection 3.2), the ranking takes into consideration both the purity information as well as the other information about each loop.

Examples Figure 4 shows example idioms, patterns mined after coiling, and concrete loops they match. Showing idioms, and not merely coiled code, allows us to illustrate both simultaneously. Loop idioms are simply a ranked selection of segments of coiled code. Map and reduce operations are quite common in our corpus. We focus at the most complex idiom in Figure 4.8 to explain the notation. The idiom contains the `<` operator, because our expression abstraction, discussed above, preserves the top-level operator in termination expressions. `INC` denotes the special node for increment expression. It contains a single block that, in turn, contains a single region that references at least (since we merge references with identical sets of nodes) four variables: 0, 1, 2, and 4. The first two are read-only unitary variables (denoted by `UR`); 2 is a collection with a read-only spine and elements (denoted by $C^S R$ for the spine and $C^E R$ for the elements); and 4 is a read-write unitary variable (denoted as URW). The reader may appreciate some of the semantic details recorded within the idioms. For example, the idiom in Figure 4.7 performs a map operation but modifies the original collection elements. It is also common in our data that loops perform multiple operations, *e.g.* idiom in Figure 4.6 is a reduce operation in `h` and a map on `d` (the code generates the Householder vector for matrix factorization in mathnet-numerics). As we will discuss in

Section 6, this idiom is one of the a common loop idioms that does not have an efficient functional replacement.

4. Descriptive Statistics

Although the methods we described in the previous section are generalizable, in this work we focus on loops. Loops represent a core and hard-to-analyze code construct. In this section, we present an empirical study of loops and LINQ statements on a large corpus of 25.4 MLOC. We show that loops and LINQ statements present some notion of *naturalness* (Hindle et al. 2012) and therefore we can exploit this fact to mine loop idioms that tool developers and language designers can use in the next sections. This study of loops contributes to a long-standing line of research. Knuth (1971) analyzed 7,933 FORTRAN programs, finding that loops, are mostly simple: 95% of them increment their index by one, 87% of the loops have no more than 5 statements and only 53% of the loops are singly nested. Here, we find comparable results but find that loop nesting is much more rare. Changes in coding practices, notably the dominance of structured programming and object-orientation, may account for this difference. Most recently, CLAPP (Fratantonio et al. 2015) studied 4 million Android loops, seeking to characterize execution time. In contrast, our focus is on the naturalness of the loop constructs. Nonetheless, both studies find similar proportions of simple loops.

Dataset Collection We collect our data from GitHub. Using the GitHub Archive, we compile a list of all C# projects. We then compute the z score of the number of forks and watchers and sum them. We use this number as each project's popularity score. For our descriptive statistics analysis (Section 4) we use the top 500 projects. Our corpus contains 277,456 loops and 1,109,824 LINQ operations.

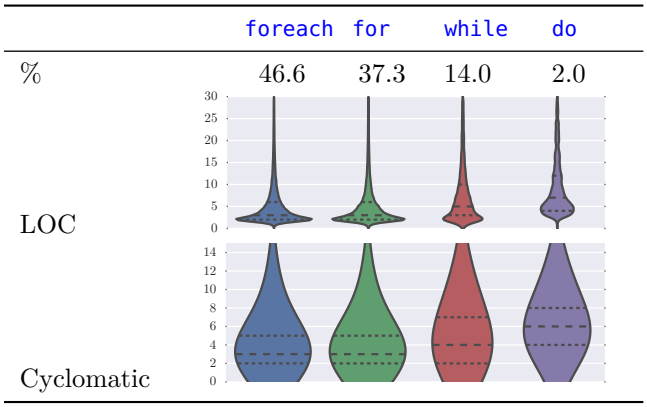
Loop Statistics We begin with some descriptive statistics over the top 500 projects that contain 277,456 loops within. Table 1 presents summary statistics for different types of loops, their sizes and complexities. The top row shows that `foreach` loops are the most popular. These `foreach` loops already represent a degree of abstraction, and their popularity suggests that programmers are eager to embrace it when available. The other loops (`for`, `while`) are less frequent, and `do` is relatively rare. The width of the violin plot gives an indication of the proportion of the sample which lies in that value range. The `foreach`, `for`, `while` loops are most often around 2 lines long, while `do` loops are a bit larger at the mode, around 5 lines. Cyclomatic complexity (McCabe 1976) measures the number of independent paths, and is used as a measure of code complexity; in our sample it is most often around 3 for `foreach` and `for`, and around 4 for `while`; this indicates that developers pack a conditional inside a short loop. `do` loops' complexities are often a bit higher, presumably because they tend to be longer. These results show that loops are natural, *i.e.* simple and repetitive. This is the key finding on which our entire loop mining technique rests. It means that patterns that are repetitive enough to be worth replacing can be efficiently and effectively found.

Further characteristics of the loops are presented in the top of Table 2. Leftmost, we see the nesting level of loops. The vast majority (90%) of the loops are singly nested; virtually all (99%) are at most 2 levels of nesting. In our corpus, virtually none at 3 levels of nesting. The second plot is the size of the loops, in LOC (we remove empty lines, comments and lines that contain only braces); 90% are under 15 LOC, and 99% under 58 LOC. The third plot shows the proportion of lines in code that are loops. On average, 4.6% of lines belong in a loop and 90% of the code has no more than 18% of loop density (*i.e.* the proportion of non-empty lines of code that are contained within loops). Finally, at rightmost we have the density of LINQ statements per kLOC in our corpus. We find that in most cases (90%) there are no LINQ constructs at all; and fully 99% of our samples have fewer than 25 LINQ statements per kLOC. These statistics and their associated graphs in Table 2 verify that most loops are simple and the Zipfian nature of their characteristics. Knowing this fact, helps us design and select models. Concretely, we used this fact to guide our selection of the Pittman-Yor process as described in Section 2

Loops per Topic To get a sense of how loops occur across domains, we used topic analysis. To extract the topics of the source code, we parsed all C# files in our 25.4MLOC corpus collecting all identifiers, except from those in `using` statements. We then split the identifiers on camelcase and on underscores, lowercasing all subtokens. We remove any numerals. Thus, for the topic model each file consists of a set of subtokens. We use MALLET (McCallum 2002) to train LDA and classify each file. For training, we used an asymmetric Dirichlet prior and hyperparameter optimization. Once topics are extracted, to analyze loops by topic, we rank the topics by different descriptive statistics.

These ordered lists (Table 2) offer a more qualitative look at the above statistics, giving some insights into the prevalence of domain-specific loop characteristics. Thus for example, the leftmost column suggests that loops in MVC (Model-View-Controller) settings tend to be very shallow in nesting, whereas loops in mathematical domains can be deeply nested (*c.f.* tensor operations). On the second column, we see topics ordered by size (LOC) of topical loops: testing loops are quite small, whereas loops relating to serialization are quite long (presumably

Table 1: Loop statistics per type. The statistical significant differences ($p < 10^{-5}$) among the loops are: (a) average LOC `do` > `while` > `for` > `foreach` and for cyclomatic complexity `do` > `while` > `foreach`, `for`.



serializing complex container data structures within a loop).

The third column shows the loop density per topic. Security concerns, native memory, testing and GUI have few loops in the code. On the other hand, code that is concerned with collections, serialization, math, streams, and buffers contains a statistically significant larger proportion of code within loops. In the last column, we present topics ordered by frequency of LINQ operator usage. We can see that LINQ operators are frequently used within session handling and testing, while it is more infrequently used for security, native memory handling, GUI and graphics.

Discussion The descriptive statistics show that loops are "natural" in that they are mostly simple and short, yet still have a long tail of highly diverse loops. This suggests that it *is* possible to mine loop idioms that can cover a large proportion of the loops, a fact that we exploit to show the utility of loop idioms in the next section. Additionally we find that loop characteristics and usage differ significantly across domains, suggesting that different patterns of loops are prevalent in different domains. Therefore, data-driven loop idiom mining is needed to uncover domain-specific loop idioms, that might be missed by humans relying solely on intuition for finding common patterns.

LINQ Operator Usage Programming rests on iteration. Loops and LINQ expressions are two ways to express iteration in code. Thus, LINQ expressions are another data source on how humans think about iteration. Patterns of LINQ expressions is a strong indicator of patterns of semantically equivalent loops. For example, if we see a Map-Reduce LINQ statement, we should expect a similar loop idiom. But if we do not see a GroupBy-Map often in LINQ we do not expect to see this in loops either. Figure 5 shows the

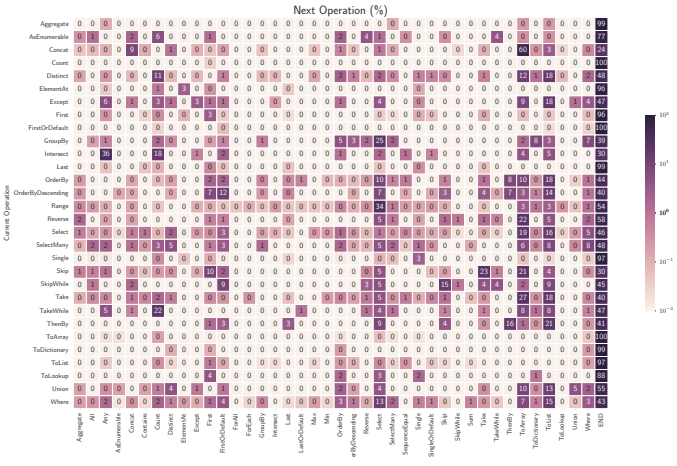


Figure 5: Common pairs of LINQ operations in our corpus. The darker the color the more often the pair is used. Numbers show the percent of times that the current operation is followed by another. The last column suggests that the current operation is the last one. Data collected across 132,140 LINQ sequences from our corpus. Best viewed in screen. A larger version can be found in Appendix B

probabilities of a bigram-like model of LINQ operations. The data is presented as a table, showing essentially transition frequencies from one LINQ operator to the next. The darker the cell, the more frequent the indicated transition. The special `END` token denotes that no LINQ operation follows. For example, a common use of `Select` is to map data from a container into another container data structure; hence `ToArray` (19% of times) or `ToList` frequently follow `Select`. In one direction, this suggests new LINQ operators; in the other, it identifies common operations that we expect to find in loops.

5. Evaluation

This work rests on the claim that we can mine semantic idioms of code to provide data-driven knowledge to refactoring tool developers and API and language designers. The goal of mining loop idioms, that represent commonly reused loop constructs, is to reduce the cost of identifying loop rewritings by working on loop idioms, instead of concrete loops. For example, a loop idiom may represent a common use of a pattern that can be simplified when introducing a new API or language feature, or an idiom can represent the LHR of a rewriting rule (*e.g.* a loop-to-LINQ refactoring) that matches a common pattern. A necessary condition for this is an effective procedure for mining loop idioms that cover, or match, a substantial proportion of real world loops. We evaluate this in this section.

Coiling Loops To coil ASTs, we need to instrument for purity (subsection 3.1) and thus need to be able to compile and run unit tests. From the top 500 projects

Table 2: Loop and LINQ statistics for the top 500 C# GitHub projects (25.4MLOC). Top high and low topics have a statistical significant difference ($p < 10^{-3}$) using a Welsh t test for the first two columns and the z test for population proportions for the other two. A full list of the topic ranking can be found in ??.

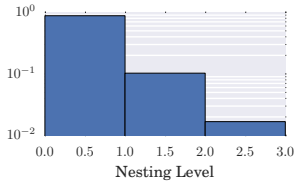
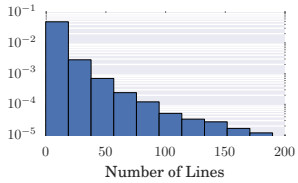
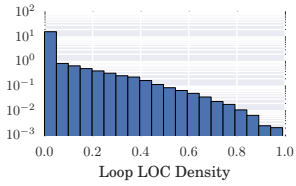
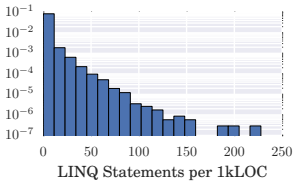
	Nesting Level	Loop Body Size (LOC)	Loop Density (% LOC)	LINQ Statements/kLOC
	 $\bar{x} = 0.14$ $p_{90\%} = 1$, $p_{99\%} = 2$	 $\bar{x} = 7.7$ $p_{90\%} = 15$, $p_{99\%} = 58$	 $\bar{x} = 4.6\%$ $p_{90\%} = 18\%$, $p_{99\%} = 56\%$	 $\bar{x} = 1.10$ $p_{90\%} = 0$, $p_{99\%} = 25$
Lowest	MVC/Events	Testing	Security	Security
	Error Handling	Native Memory	Native Memory	Native Memory
	Web/HTTP	MVC/Events	Testing	GUI
	Time/Scheduling	Collections	GUI	Graphics
	Session Handling	Session Handling	MVC/Events	Streams/Buffers
Highest	Databases	Code Manipulation	Collections	Files
	Testing	XML	Code Manipulation	Reflection
	Streams/Buffers	Streams/Buffers	Serialization	Serialization
	Graphics	Web/HTTP	Math/Temporaries	Testing
	Math/Temporaries	Serialization	Streams/Buffers	Session Handling

Table 3: C# Projects (577kLOC) from GitHub that were used to mine loop idioms after collecting purity information by running their test suite (containing 34,637 runnable unit tests).

Project	Git SHA	Description
Core	3b9517	Castle Framework Core
csvhelper	7c63dc	Read/write CSV files
dotliquid	9930ea	Template language
libgit2sharp	f4a600	C# Git implementation
log4net	782e82	Logging framework
lucenenet	70ba37	Full-text search engine
mathnet-numerics	f18e96	Math library
metrics-net	9b46ba	Metrics Framework
mongo-csharp-driver	6f237b	Database driver
Nustache	23f9cc	Logic-less templates
Sandra.Snow	c75320	Static Site Generator

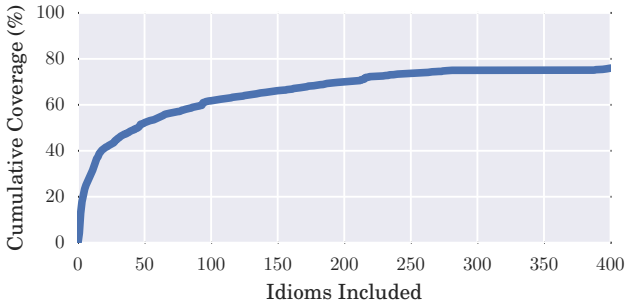


Figure 6: Cumulative loop coverage *vs.* the number of (top) idioms used. Given the diminishing returns the distributions fits well into a Pareto distribution. The Gini coefficient is $G = 0.785$ indicating a high coverage inequality among idioms. When using 50 idioms, 50% of the loops can be covered and with 200 idioms 70% of the loops are covered. 22% of the loops in our corpus are non-idiomatic (*i.e.* are not covered by an idiom).

(Section 4), we sampled 30 projects uniformly at random. We then removed projects that we cannot compile, do not have a NUnit (2016) test suite or the test suite does not have any passing tests[3], or the projects cannot depend on the .NET 4.5 framework (*e.g.* Mono projects) that is needed for our dynamic purity analysis. We end up with 11 projects (Table 3). Most of the projects are large representing a corpus of 577kLOC, containing 34,637 runnable unit tests. We executed the test suite and retrieved purity information for 5,548 methods.

Coverage of Idiomatic Loops Our idioms are mined from a large set of projects consisting of 577kLOCs (Table 3), which form our *"training corpus"*. Figure 6 shows the percent coverage achieved by the *ranked* list of idioms. With the first 10 idioms, 30% of the loops are covered, while with 100 idioms 62% of the loops are covered. This shows that idioms have a Pareto distribution — a core property of natural code — with a very few common idioms and a long tail of less common ones. This shows a useful property of the idioms. As a tool developer or a language or API feature designer uses the ranked list of idioms, she will be capturing the most useful loops but with diminishing returns as she goes down the list. In our case, the top 50 idioms capture about 50% of the loops, while the 150 idioms increases the coverage only by another 20%. Therefore,

[3] This may happen when the test suite needs an external service *e.g.* a SQL or Redis server.

our data-driven approach allows the prioritization of semantic code idioms and helps to achieve the highest possible coverage with the minimum possible effort.

Nonidiomatic Loops Figure 6 shows that about 22.4% of the loops are not covered by any of the idioms. Here, we perform a case study of these nonidiomatic loops. We sampled uniformly at random 100 loops that were not covered by any of the mined idioms and studied how they differed from idiomatic loops. We found that 41% of these loops were in test suites, while another 8% of the nonidiomatic loops were loops that were either automatically generated or were semi-automatically translated from other languages (*e.g.* Java and C). Another 13% of these loops were extremely domain-specific loops (*e.g.* compression algorithms, advanced math operations). The rest of the nonidiomatic loops were seemingly normal. However, we noticed they often contain rare combinations of control statements (*e.g.* a `for` with an `if` and another loop inside the `else` statement), convoluted control flow in the body of the loops or rare purity properties. Some of these rare combinations, like two consecutive `if`-`else` statements, are, in isolation, normal or frequent, but rare when enclosed in a loop rather than a method. We speculate these loops look normal to developers because we humans tend to notice the local normality, while neglecting the abnormality of the neighborhood. Unsurprisingly, our method also considers loops with empty bodies nonidiomatic. Knowing which loops are nonidiomatic and that they are rare is crucial, since it allows toolmakers to avoid wasting time on them.

Project Specificity of Loop Idioms So far we discussed idiom coverage with regards to the corpus used for inferring the pTSG. Now, we are interested in characterizing the project specificity of the mined loop idioms. For each of the 11 projects, we infer a pTSG on the other 10 projects and compute the coverage of the new top 200 computed idioms on the target project. The average percent of loops that are covered by the top 200 idioms trained on all the projects is 70.1% but when the project is excluded, it drops to 66.3%. This shows both that the top ranked loop idioms are general and that there is nontrivial proportion of domain-specific loop idioms. There are two exceptions: the lucenenet text search engine and the mathnet-numerics math library that about 18% of the loops are project-specific and cannot be covered by idioms found in the other projects. By manually investigating the project-specific idioms, we find that mathnet-numerics has a significant number of specialized math-related loop idioms, while lucenenet project-specific idioms are mostly in its Snowball stemmer, which is autogenerated code that has been ported from Java and is highly specialized to the text-processing domain.

These results show how mining loop idioms identify not only universal, domain-independent idioms that are frequent yet detailed enough for replacement but also domain, even project, specific loop idioms that may still benefit from custom-defined replacement transformations. Our data-driven approach allows tool developers and language designers to abandon the straitjacket of building "one size fits all" tools, forced on them by limited engineering resources, and build bespoke, even adaptable, transformation tools that many more developers can use.

Alternatives to Idiom Mining Using existing code to validate the commonality of perceived patterns is not uncommon (Okur et al. 2014; Gyori et al. 2013). However, existing tools are usually limited to simple text or pattern matching (*e.g.* `grep`). Our work differs in two important aspects. First, our CASTs provide an abstract, but semantically expressive form that can be useful on its own for matching patterns, as we show in Section 6. Second, contrary to existing tools, which provide information about a given pattern but require their user to already know the pattern, idiom mining *learns* the common idioms directly from the data without any need for a priori intuition about the patterns.

6. Using Semantic Idioms

Popular idioms identify opportunities for identifying "natural" rewritings of code, those that are structurally similar and frequent enough to warrant the cost of abstracting and reusing its core. Therefore, highly ranked idioms can suggest a new language construct, a new API call or the lefthand side of a rewriting rule that implements a refactoring. For each idiom, one has to write the righthand side of the rewriting rule. For example, loop idioms, our focus in this work, are well-suited for identifying opportunities for the task of rewriting loops using new APIs, new language constructs or even functionalizing them into LINQ statements. Because these rules are mined from actual usage, we refer to this process as *prospecting*. In this section, we discuss three case studies on using loop idioms for prospecting. In the first case study, we discuss how loop idioms can be used for prospecting loop-to-LINQ rewritings, then we show some evidence that loop idioms can help with designing better APIs or even provide data-driven arguments for introducing new language features.

6.1 Prospecting Loop to LINQ Refactorings

Loop idioms can help in an important instance of refactoring: replacing loops with functional operators. Since 2007, C# supports LINQ (Meijer 2011; Marguerie et al. 2008), that provides functional-style operations, such as map-reduce, on streams of elements and is widely used in C# code. LINQ is concise and supports lazy

operations that are often easy to parallelize. For example, multiplying all elements of the collection `y` by two and removing those less than 1, in parallel, is `y.AsParallel().Where(x=>x<1).Select(x=>2*x)`. We call a loop that can be replaced with a LINQ operator *LINQable*. LINQability has important implications for the maintainability and comprehensibility of code. LINQ's more conceptually abstract syntax 1) manifests intent, making loops easier to understand and more amenable to automated reasoning and 2) saves space, in terms of keystrokes, as a crude measure of effort to compose and read code.

As a testament to the importance of refactoring loops to functional operators, two tools already support such operations: LambdaFicator (Gyori et al. 2013) targets Java's `Streams` and JetBrain's Resharper (JetBrains 2015) replaces loops with LINQ statements. Both these tools have followed the classic development model of refactoring tools: they support rewritings that its tool developers decided to support from first principles by first deciding upon a set of preconditions, possibly verifying her intuition about which constructs are most common, and using textual matching technique. In contrast, our approach complements the intuition of the tool makers and finds important patterns that a designer may not even be aware of. Therefore, it allows toolmakers to support refactorings that the tool authors would not envision without data, enabling the data-driven, inference-based, general or domain-specific development of refactorings. Additionally, data-driven inference allows to discover project or domain-specific semantic idioms without needing a deep knowledge of a domain or a specific project.

To do this tool developers can build a refactoring tool using loop idioms as key elements to the rewritings that map loops to LINQ statements. In other words, we can use our pTSG inference to automatically identify loop constructs that could be replaced by a LINQ operator, *i.e.* are LINQ-able. In our corpus, at least 55% of all loops are LINQable.

To evaluate the fitness of our loop idiom mining for prospecting natural loop rewritings, we built an idiom-to-LINQ suggestion engine. The suggestion engine is not intended as a refactoring tool for actual developers. Instead, it is a proof-of-concept to demonstrate how loop idioms can be used by tool developers to easily build new refactoring tools, and also to demonstrate that the loop idioms have sufficient quality and convey sufficient semantic information to support the construction of practical program rewriting tools. Its suggestions are *not* sound, since it simply matches an idiom to a concrete loop and does not check any semantics-preservation semantics automating the suggested replacement would entail. For example, our idiom-to-LINQ suggestion engine maps the idiom in Figure 4.8 to a reduce operation. Thus, for the concrete loop in Figure 4.8, the suggestion engine outputs the loop and its location, then replaces references with the concrete loop's variable names and outputs the following suggestion:

```
The loop is a reduce on max. Consider replacing it with
   data.Where(cond).Aggregate((elt, max)=>accum)
1. Where(cond) may not be necessary
2. Replace Aggregate with Min or Max if possible
```

We know that this loop is a reduce because the matching idiom's purity information tells us that there is a read-write only on a unitary variable. When our suggestion engine accurately suggests a loop refactoring, a refactoring tool developer should find it easy to formalize a rewriting rule (*e.g.* identifying and checking the relevant preconditions) using loop idioms as a basis. In our example, a polished refactoring tool should refactor the loop in Figure 4.8 into `data.Where(x=>!float.IsNan(x)).Max()`.

We used the top 25 idioms that cover 45.4% of the loops in our corpus. We mapped 23 idioms, excluding 2 of the loop idioms (both `while` idioms, covering 1.5% of the loops) have no corresponding LINQ expression. To map each idiom to an expression, we found the variables that match the references, along with the purity and type information of each variable. We then wrote C# code to generate a suggestion template, as previously described. The process of mapping the top 23 idioms to LINQ took less than 12 hours.

With this map, our engine suggests LINQ replacements for 5,150 loops. Each idiom matches one or more loops and is mapped to a LINQ expression in our idiom-to-LINQ map. To validate the quality of these suggestions, we uniformly sampled 150 loops and their associated suggestions. For each of these loops, two authors assessed our engine's suggestion accuracy. This should *not* be seen as an effort for a batch-refactoring tool, but rather as an means of evaluating our proposed method. Our results show that the suggestions are correct 89% of the time. The inter-rater agreement was $\kappa = 0.81$ (*i.e.* agreed 96% of the time). So not only is our idiom to LINQ map easy to build, it also achieves good precision. This suggests that the mined idioms indeed learn semantic loop patterns that can be used for refactoring Table 4 shows the percent of loops matched by an idiom whose LINQ expression uses the specified LINQ operator and explains the most common LINQ operations. This evaluation indicates that a refactoring tool developer can easily use a loop idiom as the lefthand side of a refactoring rule. She can then write extra code that checks for the correctness of the refactoring. Most importantly, this process allows the prioritized consideration of rewritings that can provide the maximum codebase coverage with the minimal effort.

Table 4: Basic LINQ operators and coverage statistics from the top 100 loop idioms. # Idioms is the number of idioms our suggestion engine maps to a LINQ expression that uses each LINQ operator. Use frequency is the proportion of concrete loops that when converted to LINQ use the given LINQ operator.

Operator	Description	# Idioms	Use Freq
`Range`	Returns integer sequence	50	77%
`Select`	Maps a lambda to each element	42	32%
`Aggregate`	Reduce elements into a value	43	21%
`SelectMany`	Flattens collection and maps lambda to each element	5	10%
`Where`	Filters elements	13	7%
`Zip`	Combines two enumerables	6	3%
`First`	Returns the first element	2	1%

6.2 Prospecting for New APIs

The top mined loop idioms are interesting semantic patterns of the usage of code. However, some of the common patterns may be hard to read and cumbersome to write. Since semantic idioms represent common operations, they implicitly suggest new APIs that can simplify how developers invoke some operation. Thus, the data-driven knowledge that can be extracted from semantic idiom mining can be used to drive changes in libraries, by introducing new API features that simplify common usage scenarios. Due to space limitation, we present only two examples in this section.

One common set of loop idioms (covering 13.7% of the loops) have the form

```
foreach (var element in collection)
  obj.DoAction(foo(element))
```

where each element in the `collection` is mapped using `foo` and then some non-pure action is performed on `obj`. The frequent usage of this loop idiom for an API provides strong indication that a new API feature should be added. For example in lucenenet the following (slightly abstracted) loop appears

```
for (int i = 0; i < numDocs; i++) {
   Document doc = foo(i);
   writer.AddDocument(doc);
}
```

In this example, the method `AddDocument` does not support any operation that adds more than one object at a time. This forces the developers of the project to consistently write loops that perform this operation. Adding an API method `AddDocuments`, that accepts enumerables would lead to simpler, more readable and more concise code:

```
writer.AddDocuments(collection.Select(d=>foo(d)))
```

We find similar issues in other libraries, such as in mathnet-numerics where the same operation (*e.g.* a test for a specific condition) is applied in all entries of a matrix using multiple loops. For example, in the testing code of mathnet-numerics there are 717 doubly nested `for` loops that test a simple property of each element in a 2d-array. Adding a new API that accepts a lambda for each location `i,j` would greatly simplify this code.

6.3 Prospecting for New Language Features

Semantic loop idioms can provide data-driven evidence for the introduction of new language features. For example, some of the top idioms suggest novel language features. For example, five top loop idioms with total coverage 12% have the form:

```
for (int i=0; i < collection.Length; i++)
  foo(i, collection[i])
```

where they are iterating over a collection but also require the index of the current element.

A potential new feature would be the introduction of an `Enumerate` operation that would jointly return the index and the element of a collection. This resembles the `enumerate` function that Python already has and Ruby's `each_with_index`. Interestingly, using loop idioms we have identified a common problem faced by C# developers: in StackOverflow there is a related question for C# (StackOverflow 2009a) with about 93k views and a highly voted answers (195 votes) that suggests a helper method for bypassing the lack of such a function.

Prospecting for New LINQ Operators Mined loop idioms can implicitly help with designing new LINQ operators. For example, while mapping loop idioms to LINQ, we found 5 idioms (total coverage of 5.4%) that map to the rather cumbersome LINQ statement

```
Range(0, M).SelectMany(i => Range(0, N)
   .Select(j => foo(i, j)))
```

These idioms essentially are doubly nested `for` loops that perform some operation for each `i` and `j`. This suggests that a 2-d `Range` LINQ operator would be useful and would cover about 5.4% of the loops. In contrast, our data suggests that a n-d ($n > 2$) `Range` operator would be used very rarely and therefore no such operator needs to be added. We note that we have found two StackOverflow questions (StackOverflow 2010, 2013) with 15k views that are looking for a similar functionality. Another example is a set of idioms (coverage 6.6%) that map to

```
Range(M, N).Select(i=>foo(collection[i]))
```

essentially requiring a slice of an ordered collection[4]. The common appearance of this idiom in 6.6% of the loops provides strong data-driven evidence that

[4] This could also be mapped to the equally ugly `collection.Skip(M).Take(N-M).Select(foo)`.

a new feature would be highly profitable to introduce. For example, to remove these loops or their cumber -some LINQ equivalent, we could introduce a new `Slice` feature that allows the more idiomatic `collection.Slice(M, N).Select(foo)`. Indeed, the data has helped us identify a frequently requested functionality: This operation seems to be common enough that .NET 3.0 introduced the `slice` method, but only for arrays. Additionally, the need of such a feature — that we automatically identified through data — can be verified by the existence of a highly voted StackOverflow question (StackOverflow 2009b) with 106k views and 17 answers (with 422 votes in total) asking about slicing with some of the answers suggesting a `Slice` LINQ extension function.

Finally, we observe that some loops perform more than one impure operation (*e.g.* adding elements to two collections), while efficiently reusing intermediate results. To refactor this with LINQ statements an intermediate LINQ expression needs to be converted to an object (*e.g.* by using `ToList()`) to be consequently used in two or more other LINQ expressions, because of the laziness of LINQ operators. This is not memory efficient and may create an unneeded bottleneck when performing parallel LINQ operations. A memoization LINQ operator that can distribute the intermediate value into two or more LINQ streams, could remove such hurdles from refactoring loops into LINQ.

In our dataset, LINQ slicing seems to be a common idiom required across many projects suggesting that an addition to core LINQ API could be reasonable. In contrast, the 2d `Range` is specific to `mathnet-numerics`, suggesting that a domain-specific helper/extension LINQ operator could be introduced in that project, as we discussed earlier.

7. Related Work

Code clones (Basit and Jarzabek 2009; Kamiya et al. 2002; Kim et al. 2005) are related to idiom mining. Code clone detection using ASTs has also been studied extensively (Baxter et al. 1998; Jiang et al. 2007; Koschke et al. 2006). For a survey of clone detection methods, see Roy and Cordy (2007); Roy et al. (2009). In contrast to clone detection, code idiom mining searches for frequent, rather than maximally identical subtrees (Allamanis and Sutton 2014) (see Section 2). Additionally, code clones do not abstract over the semantic properties of code as we do in this work. Qiu et al. (2017) instrumented the Java parser to count the usage of production rules across various releases of Java, but do not automatically find meaningful patterns. Another related area is API mining (Acharya et al. 2007; Nguyen et al. 2009; Zhong et al. 2009; Wang et al. 2013). However, this problem is significantly different from idiom mining because it focuses on APIs calls, while in our work we abstract the exact method calls and additionally include semantic information (*e.g.* purity).

Our mining methods are directly applicable to rewritings, such as refactoring (Fowler 1999). The most prominent area of research on refactoring focuses on developing tools to automatically identify locations to refactor and/or perform refactorings (Binkley et al. 2005; Mens and Tourwé 2004; Dig 2011; Beyls and D'Hollander 2009; Kjolstad and Snir 2010) with tremendous impact: nearly all popular IDEs (*e.g.* Eclipse, Visual Studio, NetBeans) include refactoring support of some kind. However, existing refactoring tools are underutilized (Murphy-Hill and Black 2008). One reason may be the fact that many refactoring tools cannot handle many of the constructs (such as loops) that developers actually write. This is the problem we tackle in this work, by giving tool developers the tools they need to make data-driven decisions. Tsantalis and Chatzigeorgiou (2009) use machine learning-like methods to find opportunities to apply existing refactoring operators. In contrast to this work, we mine, rank and present idioms to refactoring tool developers as candidates for the left-hand sides (the pattern to replace) of new refactoring operators.

Multiple tools focus on loop rewritings. Relooper (Dig et al. 2009) automatically refactors loops on `Array`s into parallelized loops. Resharper (JetBrains 2015) provides refactorings to convert loops into LINQ expressions. Gyori et al. (2013) refactor Java loops to Java 8 streams, which are similar to LINQ in C#. All these works use the classic approach that rests on the tool developer's intuition — not data — to decide which rewritings to implement. For example, the tool of Gyori et al. (2013) only handles four loop types, comprising 46% of the loops that they encountered, underscoring the challenges of refactoring loops and the importance and utility of functionalizing them. Since all these tools contain hard-coded refactorings, they may miss refactoring opportunities that are project specific or have common constructs that the tool developers did not anticipate. Similarly, a study of vectorizing compilers, which rewrite sequential loops to use vector instructions, found that, while collectively the compilers successfully rewrote 83% of the benchmark loops, their individual performance ranged from 45–71% (Maleki et al. 2011). Our work is complementary to those tools and aims to complement the tool developers' intuition with data identifying and ranking idioms, patterns potentially worth replacing, in loops, including including domain or project specific idioms.

8. Conclusion

We presented a method for the unsupervised mining of semantic idioms, specifically loop idioms, from a

code corpus. By abstracting the AST and augmenting it with semantic facts like purity, we showed that idiom mining can cope with syntactic diversity to find and prioritize patterns whose replacement might improve a refactoring tool's coverage and help language and API designers. Semantic idioms can also benefit other other areas of program analysis and transformation, guiding the selection of heuristics and choice of corner cases with hard data, as in auto-vectorization (Barthe et al. 2013).

Acknowledgments

Parts of this work have been conducted while the first author was an intern at Microsoft Research, WA, USA. This work was further supported by Microsoft Research Cambridge through its PhD Scholarship Programme. Charles Sutton was supported by the Engineering and Physical Sciences Research Council [grant number EP/K024043/1].

References

M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC/FSE*, 2007.

C. C. Aggarwal and J. Han. *Frequent pattern mining*. Springer, 2014.

M. Allamanis and C. Sutton. Mining Idioms from Source Code. In *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

E. T. Barr, C. Bird, and M. Marron. Collecting a heap of shapes. In *ISSTA*, 2013.

G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. In *PPoPP*, 2013.

H. A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering*, 2009.

I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, 1998.

M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. Technical report, PeerJ Preprints, 2016.

K. Beyls and E. H. D'Hollander. Refactoring for data locality. 2009.

D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *ICSE*, 2005.

S. Cherem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Proceedings of the 16th International Conference on Compiler Construction*, 2007.

T. Cohn, P. Blunsom, and S. Goldwater. Inducing tree-substitution grammars. *Journal of Machine Learning Research*, 11, Nov 2010.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

D. Dig. A refactoring approach to parallelism. *Software, IEEE*, 2011.

D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in Java. In *OOPSLA*, 2009.

J. Fowkes and C. Sutton. A subsequence interleaving model for sequential pattern mining. *KDD*, 2016.

M. Fowler. *Refactoring: Improving the design of existing programs*. Addison-Wesley Reading, 1999.

Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. CLAPP: Characterizing loops in Android applications. In *ESEC/FSE*, 2015.

A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian data analysis*. CRC Press, 2013.

S. J. Gershman and D. M. Blei. A tutorial on Bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

A. Guida, F. Gobet, and S. Nicolas. Functional cerebral reorganization: a signature of expertise? reexamining guida, gobet, tardieu, and nicolas'(2012) two-stage framework. *Frontiers in human neuroscience*, 7:590, 2013.

A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *FSE*, 2013.

A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

JetBrains. Resharper. `http://www.jetbrains.com/resharper/`, 2015. URL `http://www.jetbrains.com/resharper/`.

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, 2007.

A. K. Joshi and Y. Schabes. Tree-adjoining grammars. In *Handbook of formal languages*. Springer, 1997.

T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 2002.

M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, 2005.

D. D. F. Kjolstad and M. Snir. Bringing the HPC programmer's IDE into the 21st century through refactoring. In *SPLASH*, 2010.

D. E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and experience*, 1(2):105–133, 1971.

R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering (WCRE)*, 2006.

I. Kuzborskij. Large-scale pattern mining of computer program source code. Master's thesis, University of Edinburgh, 2011.

P. Liang, M. I. Jordan, and D. Klein. Type-based MCMC. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2010.

S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, D. Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

F. Marguerie, S. Eichert, and J. Wooley. *LINQ in Action*. Manning, 2008.

M. Marron, D. Stefanovic, D. Kapur, and M. V. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *Languages and Compilers for Parallel Computing*, 2008.

T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

A. K. McCallum. MALLET: A Machine Learning for Language Toolkit. 2002.

E. Meijer. The world according to LINQ. *Queue*, 9(8):60, 2011.

T. Mens and T. Tourwé. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.

K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *Software, IEEE*, 25(5), 2008.

T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE*, 2009.

NUnit. NUnit: Unit testing framework for .NET. `http://nunit.org/`, 2016.

S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in c. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

M. Post and D. Gildea. Bayesian learning of a tree substitution grammar. In *Proceedings of the Association for Computational Linguistics (ACL)*, 2009.

D. Qiu, B. Li, E. T. Barr, and Z. Su. Understanding the syntactic rule usage in java. *Journal of Systems and Software*, 123:160–172, 2017.

V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, 2014.

C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen's University at Kingston, Ontario, 2007.

C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.

StackOverflow. C# foreach with index. `http://stackoverflow.com/questions/521687`, Feb. 2009a. URL `http://stackoverflow.com/questions/521687`.

StackOverflow. Array slices in C#. `http://stackoverflow.com/questions/406485`, Jan. 2009b. URL `http://stackoverflow.com/questions/406485`.

StackOverflow. Using LINQ with 2D array, Select not found. `http://stackoverflow.com/questions/3150678`, June 2010. URL `http://stackoverflow.com/questions/3150678`.

StackOverflow. How to search in 2D array by LINQ? `http://stackoverflow.com/questions/18673822`, Sept. 2013. URL `http://stackoverflow.com/questions/18673822`.

A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, 2005.

Y. W. Teh and M. I. Jordan. Hierarchical Bayesian nonparametric models with applications. In N. Hjort, C. Holmes, P. Müller, and S. Walker, editors, *Bayesian Nonparametrics: Principles and Practice*. Cambridge University Press, 2010.

N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *Software Engineering, IEEE Transactions on*, 35(3):347–367, 2009.

J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE*, 2007.

H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

A. Mined Topic Model

Table 5 shows the mined topics from the C# code. Table 6 presents the statistical differences in loop characteristics

B. LINQ Patterns

Figure 7 shows a larger and more legible version of Figure 5 in Section 4.

Table 5: The mined topic models for our descriptive statistics in Section 4. The naming of the topic (left column) is our interpretation of the topic. The right column shows the most common (sub)tokens in the code for each topic.

Name	Topic Tokens
Databases	`data table sql column row command db name value type connection string get parameter record cell set i add`
Collections	`value t i key list index string source to item get count dictionary array add collection exception enumerable hash`
XML	`node element xml attribute name type value x child tag id namespace path document system attributes string has parent`
Native Memory	`int ptr u gl entry open system single security handle point vertex v unmanaged tk type get graphics target`
Serialization	`object type serialization pack serializer json msg system value member target tuple context schema result read property polymorphic item`
Math/Temporaries	`m i p a b x c d f v vector j y r n matrix s result k`
Error Handling	`message exception error state service context system i result security log async channel trace callback not operation is event`
Code Manipulation	`expression code type node op arg token var expr statement block location visit result operator context ast variable add`
Networking/Games	`packet read id guid player spell game var i opcode of bit device channel card effect get actor client`
MVC/Events	`event view model args resource e item changed on page action get handler property control manager text context value`
Session Handling	`id query user var name entity i order session store filter by get list group context add cache to`
Web/HTTP	`request response token http context value url client instance get name status parameters uri async id result cancellation web`
GUI	`system text box label button windows forms size tab menu item add style drawing tool name controls layout strip`
Graphics	`color image point x width y size height cc position line frame rect font rectangle draw left vector to`
Streams/Buffers	`write writer reader stream line length read string buffer index offset text bytes i size to start count char`
Files	`file path name get project info settings directory string i package config is var version folder configuration format proto`
Testing	`assert test equal are var is expected result exception true not null to fact mock with should tests get`
Security	`val fields fix quick set field is get underlying id tags leg security type value no party date price`
Reflection	`type name method i property get info member is reference builder types parameter attribute definition field class var value`
Time/Scheduling	`time date task on span i var action queue next start add thread create status job scheduler to repository`

Table 6: Loop and LINQ statistics for the top 500 C# GitHub projects (25.4MLOC). The table presents the sorted (lower to higher) list of topics for each metric. This is an extended version of the last rows of Table 2. The red boxes next to each topic name signify the label of the topics that are *not* statistically different from the others. Note that statistical significance is not a transitive relation.

Nesting Level	Loop Body Size (LOC)	Loop Density (% LOC)	LINQ Statements/kLOC
(a) MVC/Events	(a) Testing	(a) Security	(a) Security
(b) Error Handling ≅ c,d,e	(b) Native Memory ≅ c,d,e	(b) Native Memory	(b) Native Memory
(c) Web/HTTP ≅ b,d,e	(c) MVC/Events ≅ b,d	(c) Testing	(c) GUI
(d) Time/Scheduling ≅ b,c,e,f,g,h	(d) Collections ≅ b,c,e	(d) GUI	(d) Graphics
(e) Session Handling ≅ b,c,d,f,g,h	(e) Session Handling ≅ b,d	(e) MVC/Events	(e) Streams/Buffers
(f) Collections ≅ d,e,g,h,i,j,n	(f) Security ≅ g,h	(f) Session Handling ≅ g	(f) Error Handling
(g) Files ≅ d,e,f,h,i,j,k,n	(g) Networking/Games ≅ f,h	(g) Error Handling ≅ f	(g) XML ≅ h,i
(h) Native Memory ≅ d,e,f,g,u,j,k,l,m,n	(h) Time/Scheduling ≅ f,g	(h) Networking/Games	(h) Networking/Games ≅ g,i
(i) GUI ≅ f,g,h,j,k,l,m,n	(i) GUI ≅ j,k,l	(i) Time/Scheduling	(i) Math/Temporaries ≅ g,h,j
(j) Networking/Games ≅ f,g,h,i,k,l,m,n	(j) Reflection ≅ i,k,l	(j) Graphics	(j) Web/HTTP ≅ i
(k) Serialization ≅ g,h,i,j,l,m,n	(k) Databases ≅ i,j,l	(k) Databases	(k) MVC/Events
(l) XML ≅ h,i,j,k,m,n	(l) Error Handling ≅ i,j,k	(l) XML	(l) Databases ≅ m
(m) Reflection ≅ h,i,j,k,l,n	(m) Files ≅ n,o	(m) Files	(m) Code Manipulation ≅ l
(n) Security ≅ f,g,h,i,j,k,l,m,o,p,q	(n) Graphics ≅ m,o	(n) Reflection ≅ o	(n) Time/Scheduling
(o) Code Manipulation ≅ n,p	(o) Math/Temporaries ≅ m,n	(o) Web/HTTP ≅ n	(o) Collections ≅ p,q,r
(q) Databases ≅ n,p,r	(q) Code Manipulation ≅ p	(q) Collections	(q) Files ≅ p,q,r
(p) Testing ≅ n,o,q,r	(p) XML ≅ q	(p) Code Manipulation	(p) Reflection ≅ o,q,r
(r) Streams/Buffers ≅ p,q	(r) Streams/Buffers	(r) Serialization	(r) Serialization ≅ o,p,q
(s) Graphics	(s) Web/HTTP	(s) Math/Temporaries	(s) Testing
(t) Math/Temporaries	(t) Serialization	(t) Streams/Buffers	(t) Session Handling

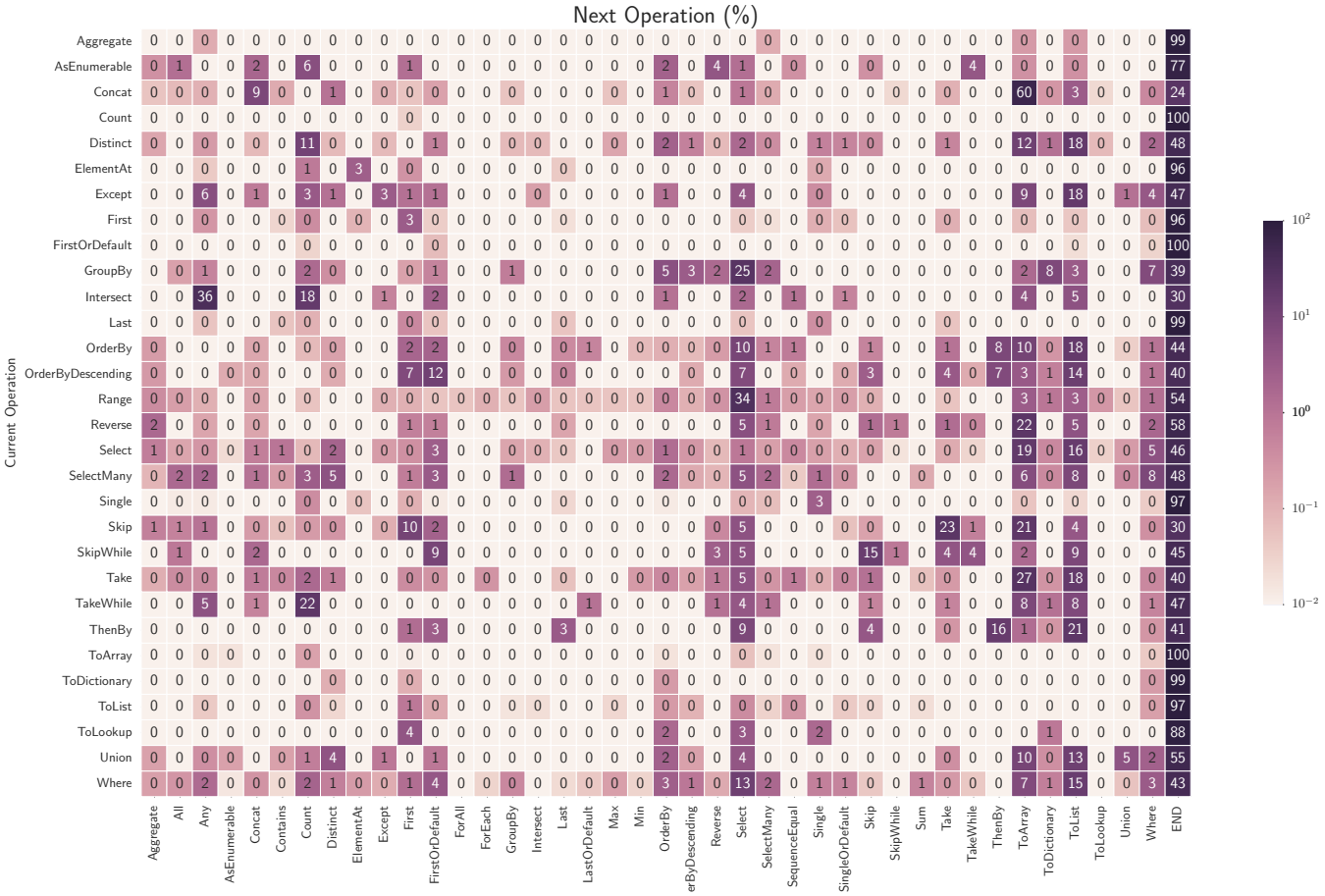


Figure 7: Common pairs of LINQ operations in our corpus. The darker the color the more often the pair is used. Numbers show the percent of times that the current operation is followed by another. The last column suggests that the current operation is the last one. Data collected across 132,140 LINQ sequences from our corpus. This is a larger version of Figure 5 in Section 4.