Sloth: Locating Sites for Repetitive Edits with Lazy Concrete Pattern Matching on Trees

Remy Wang
Rashmi Mudduluru
Hadar Greinsmark
UW CSE

ABSTRACT

Programmers sometimes need to repeat an edit over a large code base, e.g. when refactoring. Manually repeating an edit can be tedious and error-prone. We propose Sloth, a tool that locates sites that match a given template of the code to be edited. The template consists of code in the target language's (e.g. Java's) concrete syntax as well as combinators to impose complex constraints on the code to be matched. The Implemention of Sloth involved compiling the templates to Haskell patterns and matching them on the parse tree of the source code. Thanks to quasi quotation, generic programming and lazy evaluation, it took little effort (~ 200 LOC) to implement Sloth for Java.

ACM Reference Format:
Remy Wang, Rashmi Mudduluru, and Hadar Greinsmark. 2018. Sloth: Locating Sites for Repetitive Edits with Lazy Concrete Pattern Matching on Trees. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Oftentimes, programmers who develop and maintain large codebases need to apply the same code transformation at several locations. For instance, Professor Michael Ernst maintains Daikon [6], a large system for dynamic invariant detection that has over 100K+ lines of Java code. At times, Professor Ernst needs to perform repetitive edits throughout the large code base. For example, commit `11e10a` changed 57 files to change for loops into `for-each` loops[1]. Professor Ernst performed the change because `for-each` loops are more readable - a developer can be sure that every element in the traversed data collection is visited, and that the data collection remains unchanged. One example of this edit is shown in Figure 1. To convert a `for` loop into a `for-each` loop, Professor Ernst replaces the loop header `(Iterator<PptTopLevel> iPpt = ppts.iterator(); iPpt.hasNext();)` with a declaration that names the current element (`PptTopLevel ppt : ppts`). The name should match the local variable in the first line of the original body. Note that the original loop body cannot mention the iterator `iPpt` after the first line to be eligible for the edit. In addition, the loop body should not mutate the current element. Professor Ernst used the search-and-replace command in emacs with a regular expression to repeat the edit on all loops in the code base. However, since it is not obvious how to express the constraints (that the loop body cannot mention the iterator after the first line, or mutate the current element) in a regex, 14 changes triggered compiler errors or failed tests. Professor Ernst then had to revert the change on those loops one by one.

Figure 1: Example edit

```
- for ( Iterator<PptTopLevel> iPpt = ppts.iterator();
-     ; iPpt.hasNext(); ) {
-   PptTopLevel ppt = iPpt.next();
-   add (ppt);
- }
-----
+ for ( PptTopLevel ppt : ppts ) { add (ppt); }
```

Figure 2: Sloth query

```
for ( Iterator<`_> #i = `_; `_; ) {
  `_ #x = #i.next();
  `[ `! `*( #i `| #x = `_ `)* `]
}
```

Besides regular expressions, other tools are available to programmers in a similar situation as Professor Ernst's. Program transformation tools such as Stratego [2], TXL [5], and SPOON [11] operate on the parse tree and requires the programmer to understand how a specific parser parses the source. Specifically, if the programmer wishes to transform `for` loops, she has to know that a `for` loop is parsed into e.g. `for_statement [for_init] [for_expression] [for_update] [statement]` in TXL. Research tools like REFAZER [13] and LASE [8] learn repetitive edits from examples. These tools implement learning algorithms and the user provides additional examples to improve the matches if false negatives / false positives occur (neither tool claims 100% precision or recall). Our seach language can complement such tools by exposing the learned search query to the user making the tool more robust.

Stepping back, the task of repetitive edit consists of two steps: 1. identify possible code fragments to be changed (locate), and 2. perform some transformation on each eligible fragment (transform). Either step can also be repeated and interleaved with the other. In Professor Ernst's case, he first locates and transforms all loops that use an iterator with a regular expression, then locates the loops

[1] Also known as the *enhanced for.*

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Figure 3: Initial query

```
for ( Iterator<PptTopLevel> iPpt = ppts.iterator();
    ; iPpt.hasNext(); ) {
  PptTopLevel ppt = iPpt.next();
  add (ppt);
}
```

Figure 4: Partial query

```
for ( Iterator<`_> #i = `_;  `_; ) {
  `_ #x = #i.next();
  `[ `_ `]
}
```

that break the code, and finally manually transforms those loops back to original. In this paper, we focus on helping the programmer locate sites for repetitive edits.

We propose Sloth, a code search tool that allows programmers to specify the syntactic structure of their code in a language that is very close to the concrete syntax. It makes a first step towards enabling easy and accurate repetitive edits, namely to accurately specify properties of code to be edited with a simple user input. Additionally, Sloth can easily be extended to other programming languages, making it easily adoptable.

2 THE CODE SEARCH TOOL SLOTH

The following sections demonstrate the design and implementation of Sloth and its query language Yoko. We first follow a user who develops a search query from an example edit site. In this process we gradually introduce parts of Yoko's syntax and functionality. Then we present the full language design in detail. Finally, we present the architecture and implementation of the query compiler and the search engine, by each of its components.

In summary, we combine three powerful tools: quasi quotation, generic programming and lazy evaluation, to build a search interface that allows the programmer to easily find desired code fragments. Moreover, it takes little effort to implement support for new languages: given a syntax definition and a parser, it takes a total of under 250 LOC to implement Sloth for Java.

2.1 Developing a search query

To locate repetitive edit sites, the user passes Sloth the files to search over as well as the search query. In the example above, the user passes in the root of the Daikon repository and the query in Figure 2. The user may formulate the query from existing code following the steps below.

First, the user copies the original code in Figure 3 verbatim and passes to Sloth. Next, the user generalizes the query by introducing meta-variables and wild cards. The resulting partial query is in Figure 4. A meta-variable can occur in place of any parsable Java language construct - in this case the wild-cards `_ occurs in place of expressions and statements. A meta-variable preceded by # can match on identifiers that occur as different language constructs. In this case, `#i` matches on the declared iterator name in the loop header and its use in the body, and `#x` matches on the local variable name. The pattern on the last line `[`_ `]` matches on a sequence of any statement.

Finally, the user wishes to refine the loop body in the query with further restrictions. Since transforming a loop to a `for-each` loop eliminates the iterator, the transformed body cannot use the iterator in any way. In addition, since a `for-each` loop keeps the traversed data constant, the body cannot modify the data collection.

Figure 5: Search language grammar

```
p extends grammar
  := `meta-var                 // any unit
   | #meta-var                 // any identifier
   | `*( p `)*                 // any unit containing p
   | `@ | `in p                // any unit contained by p
   | `! p | p `| p | p `; p    // combinators
   | `[ p `]                   // a sequence of queries

meta-var := alpha-numeric | _
```

With this in mind, the user changes the query to assert the loop body does not use the iterator `#i` after the first line and does not assign to the current element (`#x = `_`). The negation operator `! filters out any result that matches the query following it, and the nesting operator `*( p )* matches any code containing code that matches `p`. The choose operator `p `| q` matches any code matching either `p` or `q`. In combination, `! `*(p `| q `)* filters out any code containing parts that match either `p` or `q`.

2.2 The query language Yoko

The example exercises most features of Sloth's query language Yoko. Figure 5 gives the full grammar of the language. Starting from the top, `p extends Java` means a query `p` can be any valid Java language construct, i.e. any string that can be parsed to some left-hand-side of a rule in a BNF-style Java grammar. We use the Java grammar implemented by the language-java library [4]. Although this makes Sloth grammar-dependent, the user need not know the left-hand-side of the grammar, but only needs to make sure the query does parse. We restrict the queries to match only on parsable Java constructs to avoid transformations that break the code: a transformation that replaces syntactically legal code with legal code will always result in legal programs, whereas one that replaces possibly illegal code with illegal code may not. For example, replacing a valid `while` loop with a valid `for` loop does not introduce syntax errors, but merely replacing the word `while` with `for` will. By "extend", we mean to add the following cases to every rule in the Java grammar.

Yoko extends the Java grammar first with meta-variables. A meta-variable can occur in place of any valid Java construct, and has the syntax of a back tick ` followed by an alpha-numeric string, or a wild card _. A meta-variable matches on any valid Java construct. In terms of the parse tree, a meta-variable can match on any subtree in the source. Sometimes the same identifiers can appear as different language constructs. For example, in Figure 3 `iPpt` appears both as the LHS of a declaration in the header and also as an expression in the body. We therefore introduce a different meta-variable `#i` that

matches on any identifier but ignores what construct the identifier parses to.

The nesting query `‘*( p ‘)*` where p is another query matches any code that contains a part that matches p. In terms of parse tree, it matches on any tree with a subtree that matches p.

The context query `‘in p` where p is a query matches any code that is contained by some code that matches p. In terms of parse tree, this means the matched code has an ancestor that matches p. In addition, the symbol `‘@` can occur anywhere a wild card can occur in p. In that case, `‘in p` matches any code c whose ancestor matches `p[c/@]`.

There are three logical combinators for queries: negation `‘!`, conjunction `‘;` and disjunction `‘|`. They have their usual meaning: `‘! p` matches any code that does not match p, `p ; q` matches any code that matches both p and q, and `p | q` matches any code that matches either p or q.

Finally, the sequence query `‘[ p ‘]` matches a sequence of code, each of which matches p. For Java, only statements and declarations can occur in sequences.

2.3 The search engine

As Figure 6 shows, Sloth takes as inputs the search query and source file(s) to search over. It outputs a list of matches, each of which contains a code fragment that matches the query and the fragment's location in the source. This information can be further post-processed to help the user locate the matches. Sloth provides a simple post-processor that prints out the code surrounding the matched fragments.

Internally, Sloth consists of two major components: the query compiler and the search engine. It also needs to parse the source file, but parsers for most major languages already exist[2] and it is easy to make Sloth work with existing parsers. The following sections describe each component in detail, and makes it clear the cost (in LOC) to add a new language support for Sloth.

2.3.1 Modifying the Parser. After the user inputs the query and source, Sloth first parses the source. In our implementation for Java we directly use existing Java parser [4]. There exist a vast array of parsers for different languages, so we do not consider authoring a parser as part of the effort to implement Sloth.

We slightly modify the Java parser to parse the search queries. This involves changes to 3 files: 1. We add constructors corresponding to Yoko's grammar to the data types that define the Haskell representation of the Java parse tree (syntax) (14 LOC). 2. We add the operator symbols to the lexer (12 LOC). 3. We change each parsing rule to consume the added operators and produce terms in the extended syntax (36 LOC). In total, we needed to add / change 62 LOC to obtain a parser for Yoko.

2.3.2 From Queries to Patterns. Next, Sloth translates the parsed query into a pattern match on the parsed tree from the Java source. Since patterns in Haskell are not first-class, we cannot construct a pattern from the user input at runtime. Instead we compile the parsed query to generate Template Haskell code. This adds the benefit of checking the validity of the query before it's matched: because Template Haskell is generated at Haskell's compile time, the compiler will throw an error if the query is invalid.

The acute reader may have realized: if we cannot construct Haskell patterns at runtime, shouldn't we also have to parse Yoko at compile time? Indeed! This is made possible by quasi-quotes [9], a macro-expansion mechanism that translates embedded DSL code to Haskell code. We simply define a quasi-quoter that calls the Yoko parser to produce a parse tree (Figure 7), then call the generic function `dataToPatQ` to reify the parse tree into Template Haskell patterns.

When passed `const Nothing` as first argument, `dataToPatQ` takes as second argument a query containing only Java code (i.e. no query combinators) and produces a (constant) Haskell pattern that matches exactly the code in the supplied query. We then extend `dataToPatQ` to also handle meta-variables and other query combinators. Each extension is implemented as a generic function that operate on any node in the parsed query, as follows.

2.3.3 Query Combinators. Handling meta-variable boils down to defining anti-quotations [9]. We simply supply the following pair of functions that translate meta-variables occurring in place of expressions and statements, correspondingly. The functions translate any meta-variable in the query into a Haskell binder pattern of the same name.

```
antiExpPat :: Java.Syntax.Exp -> Maybe (Q TH.Pat)
antiExpPat (MetaExp s) = Just $ varP (mkName s)
antiExpPat _ = Nothing

antiStmtPat :: Java.Syntax.Stmt -> Maybe (Q TH.Pat)
antiStmtPat (MetaStmt s) = Just $ varP (mkName s)
antiStmtPat _ = Nothing
```

Then, we add the functions as extensions to `dataToPatQ`:

```
exts = ( const Nothing  `extQ` antiExpPat
                        `extQ` antiStmtPat )
java :: QuasiQuoter
java = QuasiQuoter { ... quotePat = \str ->
                           let Right c = ...
                           in dataToPatQ exts c
                     ... }
```

After meta variables, the second simplest operator to implement is negation. Intuitively, a negated query `‘! p` would first try to match a piece of code with p. If the match succeeds, the negated query will not match on the code; otherwise it will match. The following implementation of negate takes advantage of Haskell's view pattern extension [15], which allows the programmer to call arbitrary function in a pattern and match on the result. The implementation of all of the remaining operators depend on view patterns.

```
enot :: Java.Exp -> Maybe (Q TH.Pat)
enot (ENot p) = Just
  [p|(\case { $(p_) -> False; _ -> True }) -> True|]
  where p_ = dataToPatQ exts p
enot _ = Nothing
```

[2] C parser at https://hackage.haskell.org/package/language-c,
Python parser at https://hackage.haskell.org/package/language-python
and JavaScript parser at https://hackage.haskell.org/package/language-javascript

Figure 6: Architectural Diagram of Sloth

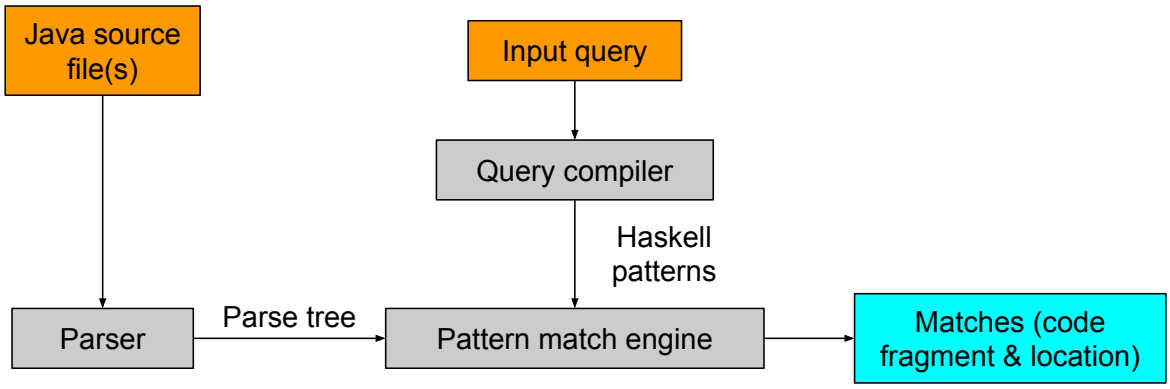

Figure 7: Quasi quotation for Yoko

```
yoko :: QuasiQuoter
yoko = QuasiQuoter {
      quoteExp = undefined
    , quotePat  = \str ->
        let Right c = parse pattern str
        in dataToPatQ (const Nothing) c
    , quoteType = undefined
    , quoteDec  = undefined
    }
```

`enot` converts any negation `ENot p` to the template Haskell pattern on the right hand side. `Just` is the identity constructor for Haskell's "option" data type `Maybe`. The quotation `[p| ... |]` reflects its enclosed Haskell code into template Haskell representation. `f -> p` is the syntax for view patterns, where f is a function and p is a pattern. In this case, p is the constant pattern `True` and f is the lambda-case `\case {$(p_) -> False; _ -> True}`, which returns False if the input matches `p_` and True otherwise. Finally, we obtain `p_` by calling `dataToPatQ` to translate the negated query p. The last line means if a pattern is not a negation, return it as-is. We implement the other two logical combinators (conjunction and disjunction) in a similar way and leave out the implementation here for conciseness.

The next interesting operator is the nesting operator `'*( p )*'`. Recall it matches on any code that contains code matching p. To implement this, we first obtain a list of all n's subtrees with the generic function `universe`. Then we use a list comprehension to select only the subtrees that match `p_`. Note that for every match, a dummy number 42 is added to the list. The actual content of the list does not matter, since we only care about whether it is empty or not. A view pattern matches the resulting list against the pattern `_:_` which asserts the list is non-empty, i.e. there is at least one subtree of n that matches p.

```
shass :: Java.Stmt -> Maybe (Q TH.Pat)
shass (SHasS p) = Just
  [p| (\n -> [ 42 | $(p_) <- universe n]) -> _:_ |]
  where p_ = dataToPatQ ext p
shass _ = Nothing
```

One last implementation detail concerns meta-variables. In a Yoko query, the same meta-variable can occur at more than one place. This is incompatible with Haskell, which require patterns to be linear, i.e. each binder can only occur once. To support non-linear queries in Yoko, we post-process the generated pattern by renaming all binders to eliminate repeated patterns, and add in guards that declare equality among all binders that used to share the same name. For example, a query `'x * 'x` compiles to the Haskell pattern `Mul x ((== x) -> True)`.

2.3.4 The matching engine. Having translated the query into a pattern match, Sloth then matches the pattern on all nodes in the parse tree. Thanks to generic programming in Haskell, we can simply implement the matching engine with a list comprehension:

```
grep :: CompilationUnit -> (Exp -> Bool) -> [Exp]
grep prog match = [ a | a <- universeBi prog, match a]

matches :: [Exp]
matches = grep prog1 match1
  where match1 [java| `! 1 |] = True
        match1 _ = False
```

The function `grep` takes a parsed source `prog` and a matching function `match`. `match` tells if an expression should be matched. On the right hand side, it creates a list comprehension. Each element in the list is drawn from the result `universeBi prog`, which is all subtrees in `prog`. Then, we select the trees matching the query by adding the guard `match a` to the comprehension. `matches` invokes grep by passing a parsed program `prog1` and the matching function `match1`, which returns true when the input matches the Yoko query inside the `[java| ... |]` quasi quote. In this case, Sloth returns all expressions that is not the literal 1.

Given the operation `[ a | a <- universeBi prog, match a]` above, the reader may be concerned about performance: if `universe` returns a list of all subtrees in n, wouldn't it be very expensive to evaluate and match each of them to the pattern p? Even worse, the nesting query also calls `universe`, so every subtree may be evaluated more than once. Thanks to Haskell's lazy evaluation [7], the operation is in fact quite cheap. Lazy evaluation only evaluates the parts of data that is used. In our case, a subtree x is only evaluated until it matches/fails to match the pattern p. For example, a 100-line `while` loop is only evaluated to `while ( unevaluated ) { unevaluated }` if it is matched against `while ( _ ) { _ }`. In short, the cost of a match is proportional to the program size and the size of matched code in terms of parse tree nodes.

3 CASE STUDIES

To evaluate Sloth, we conduct a case study on real world software. We take as benchmarks 8 sets of repetitive edits from real-world software: one set supplied by Professor Ernst from the Daikon project [6], and 7 other sets collected by [16] from Eclipse SWT. For each set of repetitive edit, we formulate queries both in Sloth and regular expression, and compare the expressiveness of the patterns based on properties of the code matched. Table 1 shows the queries in Sloth as well as in regular expression, together with comments on the regex query if it cannot match on code that contains the fragment to be edited. The following sections discuss the queries in detail, each of which demonstrates the importance of some of Sloth's design decisions or discusses possible improvements. Later, we perform a qualitative analysis of Sloth and discuss the results in the final subsection.

Q 1, 3: Only Match Parsable Code

The most important design decision in Sloth is the adoption of *concrete patterns*, i.e. search queries that use concrete Java syntax but only match on parsable code fragments. This is critical when the matched code is long and complex. In Q1, the queries should match on `if` statements that check the nullness of a field and then assign data from the field to a local variable. The regex pattern cannot match on the entire `if` statement, because regex cannot specify that two pair of braces are at the same level. Q3 raises a similar issue: to match on balanced parentheses, the regex pattern must assert there is no open parenthesis between them, therefore missing some matches.

Although sometimes matching on partial statements or expressions is sufficient for the edit, we argue that matching on parsable units helps ensure the correctness of the transformation. A transformation that replaces legal code fragments with legal code fragments is less likely to break the program than a transformation that replaces illegal code with illegal code.

Q 1, 2, 4, 6, 7: Query Combinators

Users familiar with regex may easily pick up Sloth, since the semantics of the operators closely resemble those found in regex. For example, the user may think of the nesting operator `*(p)*` as similar to the regex `.*p.*`, the sequence operator `[p]` as `p*`, and meta-variables `#x . . . #x` as character classes with capturing groups `([\w]) . . . $1`.

Q 2, 5: Ignores Comments (Shortcoming)

When formulating queries 2 & 5, we discovered an important shortcoming of Sloth. Since the Java parser that Sloth builds on ignores comments, no queries can match on comments. However, sometimes comments contain helpful information. For example, Q2 should match on all `if` statements that implement unicode traitement. Although all such statements are commented with `//-- unicode traitement`, our pattern in Sloth cannot take advantage of that information. We had to inspect a few statements to extract the common patterns among them and then encode the patterns with a Sloth query. Regex, on the other hand, very easily matches on the comments.

One possible improvement to Sloth to handle comments is as follows: once a piece of code is matched, look up the commented code from the original file and match the comments to some user supplied regular expression pattern. Then the query may look like `if (‘_) { //-*\s* unicode traitement }`.

Q 4: No Semantic Specification (Shortcoming)

The Sloth query for Q4 did not cover all changes committed to the Eclipse repository. The commit message states that the change was to refactor a local variable for an object to a field. This involves change all reference to / operations on the old local variable to refer to / operate on the field instead. These references and operations can be easily identified by a dependency analysis.

To support semantic specification such as dependency in Sloth, we may interface with off-the-shelf analysis tools and allow the user to annotate their queries with types.

Evaluation of Sloth

We evaluate the effectiveness of our tool in terms of **precision** and **recall**. We define recall as the ratio of the number of edit sites a tool matches over the number of all edit sites; we define precision as the ratio of correct matches over the number of all matches. We say that a match is *correct* when it consists of code that is edited. Each correct match should also contain only one intended edit site. Without these constraints, a trivial match that returns the entire

Table 1: Query comparison : Sloth vs regex

	Sloth query	Regex query

Q1 (Patch 1)	`if (this.#x != null) {`
	`int[] range = (int[]) this.#x.get(node);`
	`‘[ ‘_ ‘] }`
		`if (this.(\w+) != null) {\s*`
		`int[] range = (int[]) this.$1.get(node);`

Q2 (Patch 2 & 3)	`if (‘*(`
	`(( this.currentCharacter =`
	`this.source [this.currentPosition++])`
	`== ’\\’)`
	`‘)*) ‘*( (c1 = Character.getNumericValue(‘_))`
	`> 15 ‘)*}`
		`//-*\s* unicode traitement`

Q3 (Patch 4)	`switch (‘_) {`
	`case SWT.LINE_DOT:`
	`case SWT.LINE_DASH:`
	`case SWT.LINE_DASHDOT:`
	`case SWT.LINE_DASHDOTDOT:`
	`data.state &= ~LINE_STYLE; }`
		`switch ([^(]*) {\s*`
		`case SWT.LINE_DOT:\s*`
		`case SWT.LINE_DASH:\s*`
		`case SWT.LINE_DASHDOT:\s*`
		`case SWT.LINE_DASHDOTDOT:\s*`
		`data.state &= ~LINE_STYLE;\s*}`

Q4 (Patch 5)	`if (*(item)*) *(redraw)*`	`item.* redraw ([^(]*)`
Q5 (Patch 6)	`if (control instanceof Tree) { effect =` `new TreeDragAndDropEffect(‘_); } else ‘_`	`// (Drag and drop|DND) effects`

Q6 (Patch 7)	`for (int i = 0; i < digits; i++)`
	`adjustment.#x *= 10;`
	`return (int) (adjustment.#x + 0.5);`
		`for (int i = 0; i < digits; i++)`
		`adjustment.(\w+) *= 10;`
		`return (int) (adjustment.$1 + 0.5);`

Q7 (Daikon)	`for (Iterator<‘_> #i = ‘*( iterator ‘)*;`
	`#i.hasNext(); )`
	`{ ‘_ #x = #i.next();`
	`‘[ ‘! ‘( ‘*( #i ‘)* ‘| ‘*( #x = ‘_ ‘)* ‘)‘] }`
		`for (\((?>[^()]+|(?1))*\))`
		`(\{(?>[^{}]+|(?1))*\}`

Table 2: Performance of Sloth

	match locations	actual edits	Precision	Recall
Q1 (Patch 1)	8	6	0.75	1
Q2 (Patch 2 & 3)	16	24	1	0.6
Q3 (Patch 4)	15	9	0.6	1
Q4 (Patch 5)	82(22)	48	1	0.45
Q5 (Patch 6)	51(6)	6	1	1
Q6 (Patch 7)	13	10	0.7	1
Q7 (Daikon)	172	168+2+(2)	1	1

source can achieve perfect recall and precision. For instance, if the code edit was from `for(int i = 0; i < n; i++) adj.size *= 10` to `for(int i = 0; i < n; i++) adjacent.size *= 10`, and the search query looks for `adj.size` inside a `for` loop, a correct match is `for(int i = 0; i < n; i++) adj.size *= 10`. It cannot be just `adj.size`. Also, it cannot include additional statements.

We present the results of applying Sloth in Table 2. Except for Q2 and Q4, Sloth achieves a recall of 1. The reason for Q2 not achieving perfect recall is that some of the actual match locations have an `else` block following the `if` block. In order to match on all such locations, the user has to specify a query as shown in Table 1 followed by an `else` block that matches any node. This is a limitation of our current approach and we can fix this by matching on syntax tree nodes of both the types - `if` and `ifelse` whenever conditionals are involved. Q4 has low recall because we do not support dependency analysis as discussed in the previous subsection. For our experiments, we gave the entire directory of the old patch as input to Sloth. For queries Q4 and Q5, the first number under *match locations* shows the number of matches found in all files under the given directory. The number within paranthesis shows the match locations in just the files where the developer made an edit. We calculate the precision and recall using this number.

In the case of Daikon (Q7), Professor Ernst wishes to convert all eligible `for` loops into `for-each` loops. A `for` loop can only be converted to a `for-each` loop if it iterates over all elements of a data collection, and the data collection should implement the `Iterable` interface. In addition, the new loop body cannot use the iterator or modify the elements. Therefore, we use the query in the last row to match on eligible loops. `for (Iterator<‘_> #i = ‘*( iterator ‘)*; #i.hasNext(); ) { }` matches on all `for` loops that call the iterator method and access the elements of the data structure. `‘_ #x = #i.next();` ensures that the first statement in all such `for` loops must access the next element of the iterable data structure. The last part of the query, `‘[ ‘! ‘( ‘*( #i ‘)* ‘| ‘*( #x =`

`‘_ ‘)* ‘)‘]` specifies that within the `for` loop, there should be no access to the index variable and the object being iterated over should not be modified.

We ran Sloth with the query on the entire Daikon code base before the first commit related to changing to `for-each` loops. We identify eligible loops from the commit history. From the 168 loops converted to `for-each` loops, Sloth matched on all and 4 additional loops. We presented the 4 extra matches to Professor Ernst, who confirmed that two are indeed loops eligible to be converted. The other two were not changed only because an upstream repository also contains them, and changing would complicate comparison. Since the reason for not changing the last two loops does not relate to program semantics, we do not consider them as false positives.

Within the 168 loops that were changed, only 152 were changed in the first commit. One change from the first commit triggered a compiler error and was reverted in the following commit. In total, it took 5 commits spanning more than 10 years to convert all 170 loops. Using Sloth, Professor Ernst could have converted them in a single commit.

4 RELATED WORK

One of the most widely used tool for code search is regular expressions (regex), e.g. the Unix grep. The benefit of regex is its independence of language syntax, and the ability to match on malformed code as well as comments. That flexibility also implies a search-and-replace with regex may introduce syntax errors. For example, the regex `{.*}` would match `{ print '}'; }` until the first `}` in the print statement, and miss out the balanced curly brackets. Replacing the matched code with a new block will introduce parse errors (unmatched quote and unmatched closing parenthesis). Although there are extended versions of regex – like Perl's regex extension [12] – that allows the user to match on balanced matching symbols, there are other complications in the language syntax that would require the program to be parsed. Since without a parser, the user would have to be able to define a regex for any arbitrary language construct, which is equivalent to defining a parser with regex.

There are other refactoring tools that have the notion of blocks and expressions. For example, existing IDEs like Eclipse and IntelliJ can automatically discover code patterns of bad practice by applying predefined patterns of code transformations [1]. These tools all use a set of internal predefined patterns and are extremely useful for code refactoring. A search language like Sloth could be helpful for creating custom refactoring tools that can easily be written by the programmer.

Tools like REFAZER [13] (for C#) and LASE [8] (for Java) aim to automatically learn repetitive edits from edit examples. They are helpful when the learned edits reflect the user's intent. In practice, these tools work very well in that they are able to correctly learn the code transformations from the given examples. Our language can be complementary to these tools by augmenting their workflow: the learning tool can display to the user its learned template in our language, and then the user can fine tune the template to more accurately match intended code fragments.

The technique of having a pattern language for code search has been studied in [10]. This paper proposes a pattern language that extends the source language with special symbols similar to our tool. The program source code is transformed into a syntax tree and the user query is converted into a finite state automaton. As a result, this can match balanced paranthesis. This automaton takes the syntax tree as its input and generates a match if it reaches a final state. Overall, the algorithm takes $O(N^2)$ where N is the number of nodes in the syntax tree of the source code. Our tool on the other hand takes O(N) and also has the advantage of being easily extensible to other languages.

Stratego [3] is a similar tool to ours, that enables transformations with pattern matching on the parse tree. It provides a *concrete syntax* extension [14] that uses the same language as one tries to match for also writing the search queries in. By using antiquotations in the query, you create a pattern that can be matched against the target language. On the backend, the pattern gets converted to Stratego's *abstract syntax tree* format. The concrete syntax extension is limited though in that the antiquotations only can define a placeholder matching any node in the parse tree. There is no possibility to match for negations in a specific chunk. E.g. matching for a node that doesn't contain the usage of a specific variable.

5 LIMITATIONS OF OUR APPROACH

(1) While performing our experiments, we realized that matching on comments can be extremely useful sometimes. Our tool does not support this currently.
(2) Although matching on the syntactic structure of the code yields precise results, this requires the user to know the structure of the code. For instance, if a user wants to search for code inside an *if* block, but the actual code also has an *else* following it, then the user needs to specify the *else* block as part of their query.
(3) Our implementation currently does not support searching for code within a context.
(4) We also do not have support for dependence analysis. Having this would help express a richer class of queries such as searching for variables of a given type.

REFERENCES

[1] 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.* 72, 1-2 (June 2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003
[3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of computer programming* 72, 1-2 (2008), 52–70.
[4] Niklas Broberg. 2015. language-java: Manipulating Java source: abstract syntax, lexer, parser, and pretty-printer. https://hackage.haskell.org/package/language-java. (December 2015).
[5] James R. Cordy. 2006. The TXL Source Transformation Language. *Sci. Comput. Program.* 61, 3 (Aug. 2006), 190–210. https://doi.org/10.1016/j.scico.2006.04.002
[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99).* ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/302405.302467
[7] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III).* ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856
[8] John Jacobellis, Na Meng, and Miryung Kim. 2013. LASE: An Example-based Program Transformation Tool for Locating and Applying Systematic Edits. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE*

'13). IEEE Press, Piscataway, NJ, USA, 1319–1322. http://dl.acm.org/citation.cfm?id=2486788.2486995

[9] Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell '07)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1291201.1291211

[10] Santanu Paul and Atul Prakash. 1994. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering* 20, 6 (1994), 463–475.

[11] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. 2006. *Spoon: Program Analysis and Transformation in Java*. Research Report RR-5901. Inria. https://hal.inria.fr/inria-00071366

[12] Perl 5 Porters. [n. d.]. perlre - Perl regular expressions. ([n. d.]). https://perldoc.perl.org/perlre.html

[13] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

[14] Eelco Visser. 2002. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*. Springer, 299–315.

[15] P. Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 307–313. https://doi.org/10.1145/41625.41653

[16] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. 2015. Interactive Code Review for Systematic Changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 111–122. http://dl.acm.org/citation.cfm?id=2818754.2818771