A Lightweight Polyglot Code Transformation Language

AMEYA KETKAR*†, Gitar Inc, USA
DANIEL RAMOS*†, Carnegie Mellon University, and INESC-ID / IST, Universidade de Lisboa, Portugal
LAZARO CLAPP*, Gitar Inc, USA
RAJ BARIK*, Gitar Inc, USA
MURALI KRISHNA RAMANATHAN*, Amazon Web Services, USA

In today's software industry, large-scale, multi-language codebases are the norm. This brings substantial challenges in developing automated tools for code maintenance tasks such as API migration or dead code cleanup. Tool builders often find themselves caught between two less-than-ideal tooling options: (1) language-specific code rewriting tools or (2) generic, lightweight match-replace transformation tools with limited expressiveness. The former leads to tool fragmentation and a steep learning curve for each language, while the latter forces developers to create ad-hoc, throwaway scripts to handle realistic tasks.

To fill this gap, we introduce a new declarative domain-specific language (DSL) for expressing interdependent multi-language code transformations. Our key insight is that we can increase the expressiveness and applicability of lightweight match-replace tools by extending them to support for composition, ordering, and flow. We implemented an open-source tool for our language, called PolyglotPiranha, and deployed it in an industrial setting. We demonstrate its effectiveness through three case studies, where it deleted 210K lines of dead code and migrated 20K lines, across 1611 pull requests. We compare our DSL against state-of-the-art alternatives, and show that the tools we developed are faster, more concise, and easier to maintain.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: Source-code rewriting, Automated refactoring, Code cleanup

ACM Reference Format:
Ameya Ketkar, Daniel Ramos, Lazaro Clapp, Raj Barik, and Murali Krishna Ramanathan. 2024. A Lightweight Polyglot Code Transformation Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 199 (June 2024), 25 pages.
https://doi.org/10.1145/3656429

1 INTRODUCTION

Automated code transformation tools are crucial to facilitate refactoring [36], migrating code [25], fixing bugs [6], managing technical debt and enhancing codebase maintainability [53]. However, automating code transformations is challenging. Code transformations are usually a web of *cascading* and *interdependent* changes that span and propagate across multiple files or repositories [30, 60]. Moreover, in contemporary domains like mobile development these changes also span across programming languages (e.g., Android where `Java` and `Kotlin` co-exist and interoperate [7]).

*This work was done when these authors were employed at Uber Technologies, Inc.
†Equal contribution at writing.

Authors' addresses: Ameya Ketkar, ameya@gitar.co, Gitar Inc, San Mateo, California, USA; Daniel Ramos, danielr@cmu.edu, Carnegie Mellon University, and INESC-ID / IST, Universidade de Lisboa, Lisboa, Portugal; Lazaro Clapp, lazaro@gitar.co, Gitar Inc, San Mateo, California, USA; Raj Barik, raj@gitar.co, Gitar Inc, San Mateo, California, USA; Murali Krishna Ramanathan, mkraman@amazon.com, Amazon Web Services, Santa Clara, California, USA.

Frameworks for automating code transformation vary widely. At one end of the spectrum, *lightweight techniques* [5, 18, 58] offer declarative languages to rewrite code with simple match-replace rules. The key advantage of lightweight techniques is language agnosticism, which stems from the techniques being independent of the underlying compiler infrastructure. Moreover, match-replace rules are often syntactically close to the target language, making them easy to write and use [58]. However, lightweight techniques are often limited to atomic context-free code changes, lacking support for tasks requiring cascading and interdependent code changes. On the other end, *imperative frameworks* [1, 14, 16] for AST-level manipulation allow for arbitrary code transformations. They provide APIs to control where, when, and how code should be rewritten based on context, symbol information, and analyses. However, these frameworks are monuments of engineering, and demand significant time and effort to learn [32]. Moreover, imperative frameworks are often language-specific, and rely heavily on underlying compiler infrastructure. This results in an additional burden when automation must support multiple languages or versions [53].

Our key observation is that the main benefits of imperative frameworks (i.e., the ability to encode cascading and interdependent code changes, and leverage code context) can also be captured by extending the match-replace system of lightweight techniques, while keeping its strengths. Specifically, we design a Domain-Specific Language (DSL) to allow defining flow and dependencies between lightweight match-replace rules, as well as the ability to capture surrounding code context by composing and using multiple match-replace rules.

The DSL provides strategies for specifying flow and dependencies between match-replace rules using the concept of a *directed edge-labelled graph of match-replace rules*. Specifically, nodes in the graph represent individual transformations rules and the edges determine the order for applying these rules. Each edge is also associated with a label that defines the scope in which the target rule is applied with respect to the source rule. For example, an edge $\mathcal{R}_1 \xrightarrow{\text{class}} \mathcal{R}_2$ reads as, "apply rule $\mathcal{R}_1$ and then apply rule $\mathcal{R}_2$ within the enclosing class where $\mathcal{R}_1$ was applied". Individual rules are expressed by interleaving any source code matching language of choice (e.g., `tree-sitter queries` [13], `concrete syntax` [11], or `regular expressions` [22]). Furthermore, rules can compose multiple matchers and matching languages using a set of *filter* primitives. Our intuition is that we can approximate relevant symbolic information and code context for precise transformations by using (1) multiple filters for syntactic checks on the surrounding code context, (2) combining multiple matching paradigms, and (3) using multiple interdependent rules [1].

There are several benefits to our approach. Firstly, it extends the traditional lightweight match-replace system and makes it *more expressive*; in our domain specific language *cascading* code transformations are a first-class citizen. Moreover, it allows us to create precise matches using composition, even though our approach is lightweight in nature. Secondly, it inherits the *familiarity* and retains the *declarativeness* from lightweight match-replace systems [58]. Lastly, it is *polyglot*. It has native support for multi-language changes, which is hard to do with imperative frameworks.

We implemented our approach as PolyglotPiranha at `Uber`, a large software company. We showcase PolyglotPiranha's expressiveness by developing *three* complex automated code transformation tools for - (1) deleting code related to *stale feature flags*, (2) large-scale migration related to `Uber`'s new *Experimentation API*, and (3) migrating an *annotation processor*. We evaluate the effectiveness of these tools by applying them across `Uber`'s proprietary `Android` and `iOS` codebases (around 7.5M lines of code `LoC` each). PolyglotPiranha-based tools deleted 210K `LoC` of stale code and migrated 20K `LoC` of old code over 1611 Pull Requests (PRs) [27], where it automated between 73.4% and 100% of all the code changes for each use case. Further, we compare the PolyglotPiranha-based tools with real-world production level tools developed upon alternatives - *Imperative*

[1]The soundness of the transformations depends on the accuracy and comprehensiveness of the rules in the graph.

```java
class Flags {
- // Declares location
- @Value("location")
- static boolean isLocEnabled() {...}
...
}
```

(a) `Java` class declaring `isLocEnabled`.

```kotlin
class SomeClass {
 ...
 fun apply(x: Int) : Int{
  - if(isLocEnabled() || x > 0) {
     return x + 1
  - }
  - return 2
 }
}
```

(b) A `Kotlin` class `SomeClass` using `isLocEnabled`.

Fig. 1. Cascading code changes after deleting the method `isLocEnabled`, and replacing its callsites with `true`. Highlighted parts represent deleted portions of the code after the change.

frameworks (`ErrorProne` [1] and `OpenRewrite` [41]) and *lightweight frameworks* (`Comby` [58]). In particular, we show that our PolyglotPiranha-based feature flag cleanup tool is significantly faster than `Piranha` [53] (based on `ErrorProne`), by 42.5× on average, with similar accuracy. We also show that PolyglotPiranha is on average 12.32× faster than `Comby` for feature flag cleanup due to ordering of rules and their controlled application within the scope, while also being more concise than imperative variants.

In summary, our main contributions are:

(1) A new declarative language designed for automating code transformations. This language allows users to express complex, multi-language code transformations (Section 3, Section 4).
(2) An extensive evaluation of the technique, demonstrating that it can address complex real world code transformations. These tools have been applied and assessed across `Uber`'s codebase, and compared against other state-of-the-art tools (Section 5).
(3) We open-source PolyglotPiranha, our implementation of the code transformation language, as well as the tool implementations for feature flag cleanup [57].

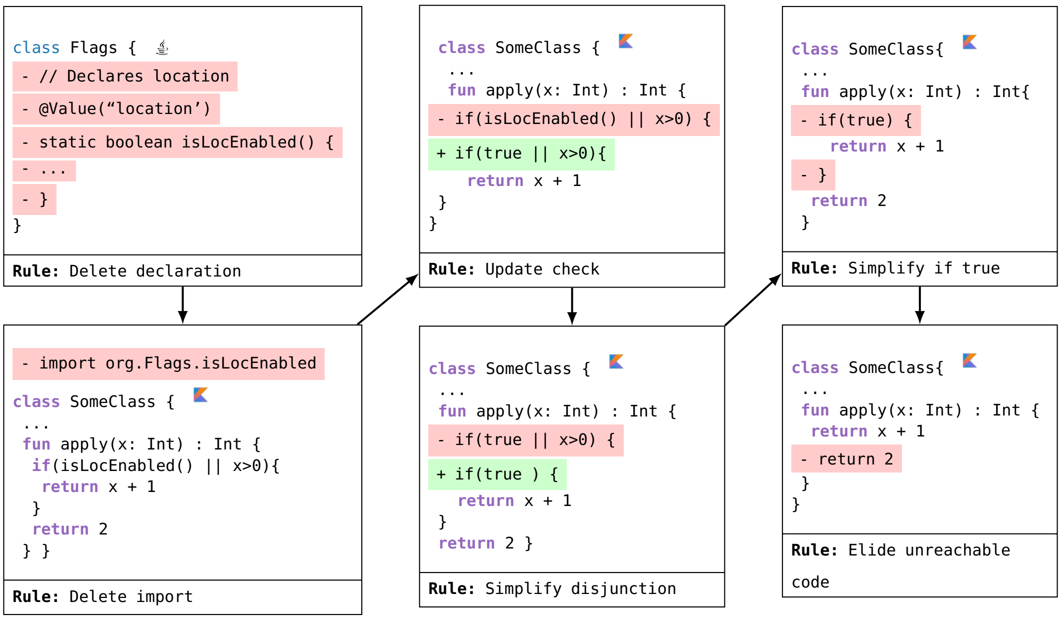
2 OVERVIEW

In this section, we provide an overview of our domain-specific language using an illustrative example of a real-world code transformation task, simplified from an automated cleanup task performed at `Uber` using PolyglotPiranha (which we explain thoroughly in Section 5.1). We show how this transformation is encoded and executed across a multi-language codebase by our tool.

Consider the code change in Figure 1a, where a developer deletes the method declaration `isLocEnabled` (annotated with `@Value("location")`) from the `Java` class `Flags`, and replaces all its usages with `true`. Figure 1b shows the `Kotlin` class `SomeClass` using the method `isLocEnabled()` inside an `if` condition. In this class `isLocEnabled` is replaced with `true`, leading to cascading code changes. Figure 2 shows the chain of code for this refactoring: (1) simplifying `true || x > 0` to `true`, (2) deleting the redundant `if(true)` statement, and (3) deleting the unreachable `return`.

To automate the code change in Figure 1, we can construct a graph of match-replace rules using our DSL (detailed in Section 3), as depicted in Figure 3. The graph has a source / seed rule `Delete declaration`, i.e., this rule instantiates and triggers the code transformations by deleting the method declaration. The rule has two components: (1) match, and (2) replace. The *`match`* is a pattern, with the holes `:[trgt]` and `:[name]`, used to signify placeholders that can match arbitrary nodes in the program's parse tree [2]. The *`replace`* indicates the replacement string. In this case, the replacement

[2]The *holes* are syntax-aware variants of named captured groups used in regular expressions [22].



```
class Flags {
- // Declares location
- @Value("location')
- static boolean isLocEnabled() {
- ...
- }
}
```
Rule: Delete declaration

```
- import org.Flags.isLocEnabled
class SomeClass {
 ...
 fun apply(x: Int) : Int {
  if(isLocEnabled() || x>0){
   return x + 1
  }
  return 2
 } }
```
Rule: Delete import

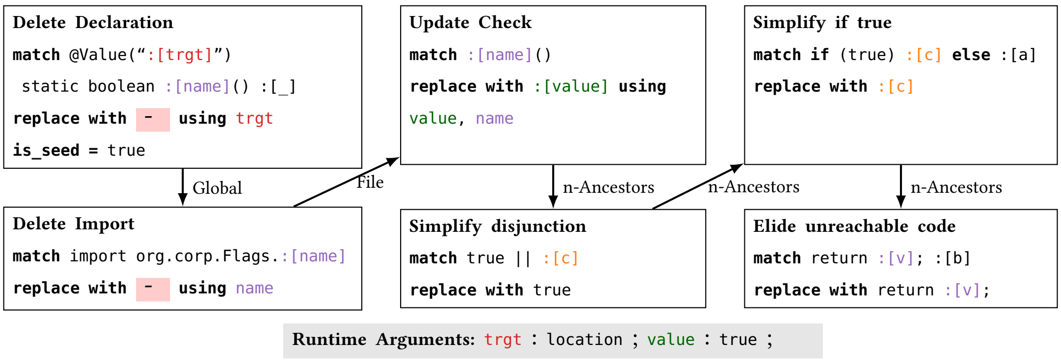
```
class SomeClass {
 ...
 fun apply(x: Int) : Int {
- if(isLocEnabled() || x>0) {
+ if(true || x>0){
    return x + 1
 }
}
```
Rule: Update check

```
class SomeClass {
 ...
 fun apply(x: Int) : Int {
- if(true || x>0) {
+ if(true ) {
   return x + 1
 }
 return 2 }
```
Rule: Simplify disjunction

```
class SomeClass{
 ...
 fun apply(x: Int) : Int{
- if(true) {
    return x + 1
- }
  return 2
 }
```
Rule: Simplify if true

```
class SomeClass{
 ...
 fun apply(x: Int) : Int {
  return x + 1
- return 2
 }
}
```
Rule: Elide unreachable code

Fig. 2. Code simplification steps after the function `isLocEnabled` is removed from the codebase, and its callsites are replaced with `true`. This transformation affects `Java` and `Kotlin` files.



Delete Declaration
```
match @Value(":[trgt]")
 static boolean :[name]() :[_]
replace with  -  using trgt
is_seed = true
```

Delete Import
```
match import org.corp.Flags.:[name]
replace with  -  using name
```

Update Check
```
match :[name]()
replace with :[value] using
value, name
```

Simplify disjunction
```
match true || :[c]
replace with true
```

Simplify if true
```
match if (true) :[c] else :[a]
replace with :[c]
```

Elide unreachable code
```
match return :[v]; :[b]
replace with return :[v];
```

Runtime Arguments: `trgt : location ; value : true ;`

Fig. 3. Program in our DSL used for cleanup described in Figure 1. The references to the runtime arguments (in the rules) are substituted with the appropriate value during execution.

is a deletion as indicated by *`replace with`* ` - `. The clause *`using trgt`* signifies that the rule is dynamic, and takes as input the variable *`trgt`*. During this transformation, the reference `:[trgt]` is substituted with the value `location` as indicated by runtime arguments. The rule matches and deletes the method "`isLocEnabled`", and `name` is bound to "`isLocEnabled`". PolyglotPiranha also detects and deletes the comment "`// Declares Location`" associated to `isLocEnabled`.

Rule graphs are explored in a *depth-first* fashion (as explained in Section 4). Each edge is labelled with a *scope*, to determine where the next rule is applied with respect to the current rule. While `Global` scope directs that the next rule should be applied everywhere in the codebase, `File` scope restricts the application of the next rule within the enclosing file where the current rule was applied, and `n-Ancestors` scope limits the next rule to the *n* ancestors of the parse node rewritten by the

```
<program> ::= <rule_graph> <substitutions>
<rule_graph> ::= <rule>+ <edge>*
<edge> ::= from string to (string, scope)
<scope> ::= Global | File | n-Ancestors
         | Method | Class
<rule> ::= name string match <match>
      [replace [template_variable] with <replace>]
      [where <filters>]
      [using <holes>]
      [is_seed bool]
      [belongs_to <groups>]
```

```
<match> ::= concrete_pattern | structural_query
        | regular_expression
<replace> ::= string <replace>
        | template_variable <replace> | <>
<filters> ::= enclosing <match> [<contains>]<filters>
        | not_enclosing <match> <filters> | <>
<contains> ::= contains <match>
             [at_least int] [at_most int]
        | not_contains <match> | <>
<holes> ::= template_variable <holes> | <>
<groups> ::= string <groups> | <>
<substitutions> ::= <>
        | template_variable value <substitutions>
```

Fig. 4. Syntax of our DSL for cascading code transformations. The elements inside square brackets are optional. The symbols `concrete_pattern`, `structural_query`, `regular_expression` are expressions for pattern matching in their respective languages (explained in Section 3); the symbol `template_variable` represents named capture groups from the matched patterns.

current rule. In Figure 3, we observe the seed rule `Delete declaration` is connected to `Delete import` with an edge labelled `Global`, therefore the latter is applied across the codebase.

To express interdependent code changes, the rules can use information from previous rule applications (like previously matched code). For instance, the rule `Delete import` takes input `name`, which is recorded in the previous rule `Delete Declaration`. Note that the value of `name` is only known at run time, since it corresponds to the name of the method which is annotated as `@Value("location")`. In the example, `name` will be instantiated with `isLocEnabled`.

As observed in Figure 2, after finishing the rule `Delete import`, PolyglotPiranha transitions to the `Update check` rule, where the usages of `isLocEnabled` within the same `File` are replaced with `true`. Notice that `Update check` also depends on the `name` of the method deleted and the replacement `value` supplied originally, as indicated by the *using* keyword in Figure 3. Finally, a sequence of cleanup rules is applied in order to simplify the code as much as possible. These sequential transformations are shown in Figure 2. Note that each rule or rule *sub*-graphs can be re-used across multiple tasks and languages. For instance, `Delete import` rule could potentially be used for a API migration and `Simplify Disjunction` could be used across `Java`, `Kotlin` and `Scala`.

3 THE CODE TRANSFORMATION LANGUAGE

Our *domain-specific language* captures complex code transformations as a graph of interleaved structural match-replace rules. The language provides off-the-shelf strategies to compose rules, propagate information between the rules, and control their application to specific scopes. In this section, we will provide a higher level explanation of the syntax of this language, and Section 4 details the run time semantics.

Figure 4 describes the grammar of our DSL. At a high level, a program in the DSL is a graph of match-replace rules. The rule graph is captured as a list of *directed* and *labelled* edges. Each node represents an individual transformation rule that structurally matches and rewrites specific code snippets. Rules can also just match code without transforming it (this is useful for e.g., finding variable names). The edges between rules specify *which* rule to apply next and the scope within which it should be applied.

3.1 Edge

As shown in Figure 4, the edges are *directed and labelled.* Each *edge* connects either two rules or a rule to a rule group, defining the order in which they should be applied, akin to the `andThen` operator[3]. The edge label specifies the *scope* of application, selecting the portion of the code base upon which the target rewrite rule is applied, with respect to the code that the source rule matched. For example, an edge from $\mathcal{R}_1 \xrightarrow{\text{method}} \mathcal{R}_2$, reads as "apply $\mathcal{R}_1$ and then apply $\mathcal{R}_2$ within the enclosing *method* where $\mathcal{R}_1$ was applied".

The DSL supports three *language-agnostic* predefined scopes as shown in Figure 4. (1) *`Global`* the target rule is applied across the codebase, (2) *`File`* the target rule is applied in the enclosing file, and (3) *`n-Ancestors`* the target rule is applied to the nearest `n` tree nodes along the path up towards the root, originating at the tree node of the code fragment that the source rule matched. It is also possible to support other *language-specific* scopes that depend on the granularity of the internal representation of code within the implementation. Our implementation, PolyglotPiranha, represents code internally with `tree-sitter` [13], and supports (3) *`Method`* the rule is applied to the enclosing method where the preceding rule was applied, and (4) *`Class`* scope refers to the enclosing class. Note that the language specific scopes are set up only once per language.

3.2 Rule

Besides the name, a match-replace rule has four major components (1) *match* - a pattern to match source code, (2) *replace* - a pattern to rewrite the matched code, (3) *filter* - to filter out certain matches based on the surrounding code, and (4) *holes* - variables referenced in the rule, filled at run time and serve as the *dynamic* component of the rule. Furthermore, a rule could be a `seed_rule`. These seed rules are entry points to the graph. This graph is traversed in a depth-first manner at each location where the rule was applied. A valid rule graph contains at least one *seed rule.*

3.2.1 Match. The *match* expression is a declarative pattern that captures a code snippet with a specific *structure* or *shape* (based on its parse tree). The match also labels portions of the matched parse tree like the *named captured groups* [22] in regular expressions. Our DSL can support multiple structural matching languages, as long as they support named capture groups (used to label portions of the code as well as tag parts that need to be replaced). Supporting multiple matching languages improves the expressiveness of PolyglotPiranha at matching code, as different languages have different strengths. PolyglotPiranha's current implementation supports three languages for its **`match`** syntax: (1) *concrete patterns*, (2) *structural queries*, and (3) *regular expressions.* Next, we detail the syntax of concrete and structural pattern matching languages supported by PolyglotPiranha.

Concrete Patterns: A concrete pattern is a string with template variables / holes, that is matched to concrete syntax nodes [2] from the program's parse tree [4]. Formally, let s be a concrete pattern containing holes of the form `:[var1]`, where each hole can represent syntactically valid sub-trees. A CST node t matches s if, traversing t in depth-first order yields leaf nodes with a string representation that aligns with s from left to right. Each hole can represent entire sub-tree structures (i.e., multiple sequential leaf nodes under an internal node). This paradigm of matching is supported by multiple other tools (e.g., [5, 58]). PolyglotPiranha adopts the syntax proposed by van Tonder and Le Goues [58] in their tool `Comby`. However, our concrete patterns have stricter semantics compared to `Comby`. In our concrete pattern, a template hole, `:[x]`, matches whole syntactic structures / CST nodes, whereas `Comby` templates can represent arbitrary strings. Figure 5 shows three examples.

[3]https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen-java.util.function.Function
[4]We use Concrete Syntax Trees (CST) over Abstract Syntax Trees (AST) because we must preserve all syntactic structures within the source code, which are necessary for source code matching

Rule	Matched code snippet	Capture Groups
`match :[m_name]()`	`isLocEnabled()`	`m_name: isLocEnabled`
`match :[mod] :[type] isLocEnabled() :[body]`	`static boolean isLocEnabled() {...}`	`mod: static type: boolean` `body: {...}`
`match import org.corp.Flags.:[name]`	`import org.corp.Flags.isLocEnabled`	`name: isLocEnabled`

Fig. 5. Example rules using concrete patterns applied across the files for the motivation example in Figure 1.

Rule	Matched code snippet	Capture Groups
`match (binary_expression (left: true) (operator: ||) right: (_) @other)`	`true || x > 0`	`other: x > 0`
`match (field_declaration (name:(identifier) @name) (rhs:(string_literal) @rhs) (#eq? @rhs "location")`	`public static String FLAG = "location"`	`name: FLAG`

Fig. 6. Examples rules using structural queries applied across the files for the motivation example in Figure 1.

Structured Query Language: A query consists of one or more patterns, where each pattern is an *s-expression* or an *xpath* that matches a certain set of nodes in a syntax tree. These queries capture the structure of the target pattern in terms of AST node types and string based predicates. This paradigm is programming language agnostic, and is supported by systems like `tree-sitter` and `JavaML`. PolyglotPiranha supports the s-expression based tree-sitter queries [12] (see Figure 6).

Each matching paradigm has distinct advantages and disadvantages. By construction structural queries are more precise than concrete syntax because they can leverage node-types or absence of particular nodes, and therefore leave less room for ambiguity (e.g., it is possible to differentiate between a field and a local variable declaration). For example, matching method declarations is easier with structural query, because we would not need to account for all its syntactic variations (e.g., modifiers like `public, static, final`) like in concrete syntax. In contrast, matching API invocation pattern like `isLocEnabled()` (from Figure 1) the *concrete pattern* is convenient and more succinct. The structural query for this pattern is verbose, and requires knowledge of the target language's grammar. Regex matching is more suitable for semi-structured documents like markdown files. Note that PolyglotPiranha is not tied to these three languages, more can be supported.

3.2.2 Replacement. The replacement pattern decides on how a matched code snippet should be transformed. It is possible to either replace the entire matched code or just segments identified by a named capture group. The replacement expression / pattern can be seen as partial function that is instantiated at run time by substituting a referenced named groups or template variables with their values from either the initial match in the rule, or inputs to the rules declared with the `using` keyword (i.e., code snippets captured in previous rule applications, or the input substitutions). Figure 7 shows three examples.

3.2.3 Filters. To make the *rules* more precise and context-aware, our DSL provides `filters` to control the application of a rule based on the surrounding code. First, the candidate code to transform is checked against the `matcher` of the rule. Then, at each matched location, the filters will check if the surrounding code of this location satisfies certain criteria.

Rule	Source Code Update
match public static String :[name] = "location" **replace** :[name] **with** SOME_:[name]	public static String ~~FLAG~~ = "location" public static String SOME_FLAG = "location"

match isLocEnabled()
replace with true
	if(~~isLocEnabled()~~)
	if(true)

match import :[q].isLocEnabled **replace with** -	- import org.corp.Flags.isLocEnabled

Fig. 7. Example replacement rules using concrete syntax.

Rule	Source Code Update

Delete unused local variable
match :[type] :[var_name] = :[rhs];
replace with -
where enclosing (method_declaration)
contains :[var_name] **atmost** 1
	void consume() {
	- int x = 10;
	execute();
	}

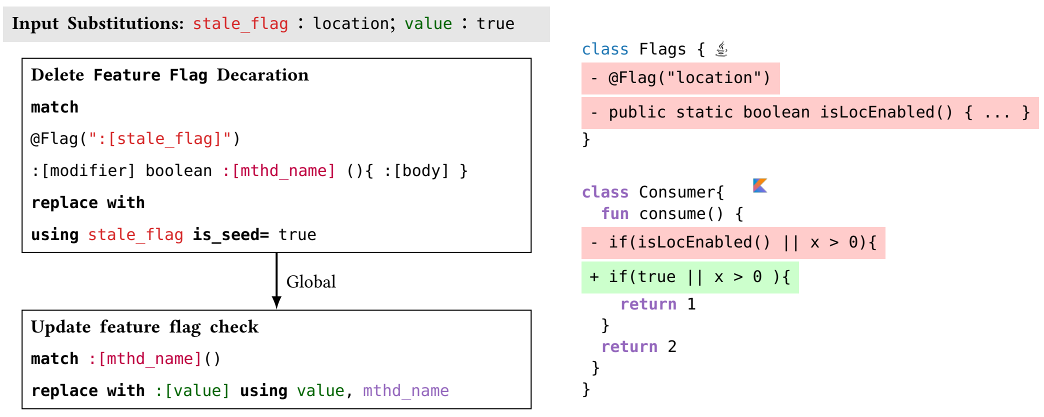
Add import statement if absent
match (package_declaration) @p
replace with :[p] \n import java.util.List;
where enclosing_node (compilation_unit)
not_contains import java.util.List
	package corp.util;
	+ import java.util.List;
	import java.util.Map;
	
	class A
	Map<String, String> m;
	List<String> l;
	}

Fig. 8. Example rules using *filters*. Note how these rules leverage both *concrete pattern* and *structural query*.

There are two primitive filters: (1) `enclosing` – checks if the primary match is enclosed by a parse tree node that satisfies the given matcher, and (2) `not_enclosing` – checks if the primary match is *not* enclosed by parse tree node that satisfies the given matcher. The `enclosing` filters can be further refined by specifying `contains` and `not_contains` expressions. The `contains` (`not_contains`) expressions specify `matchers` that should (not) match at least once inside the `enclosing_node`. The user can also specify the frequency of these matches with `at_least` and `at_most` attributes.

Figure 8 shows the rule *delete unused local variable* implemented using filters. The `match` captures the *shape* of a local variable declaration in Java, and `replace` deletes this matched code. The `enclosing` filter ensures that the primary match is inside a method declaration. It then checks if the variable name (in the primary match) matches at most once within the method. If so, the rule deletes the matched variable declaration statement (`at_most` is set to **1** to account for the variable declaration itself). While this rule does not capture all the possible scenarios of unused local variables, we found that this simple rule is very effective in practice. Similarly, the rule `Add Import statement if absent`, matches the package declaration and adds the import under it iff. it is not in the enclosing *compilation unit* (the root element of Java parse tree).

3.2.4 Holes. These serve as dynamic components within a rule. They describe input variables to the rule. At run time, their corresponding values are populated from a symbol table (which maintains the bindings from named captured groups to code snippets from current and previous



```
Input Substitutions: stale_flag : location; value : true

Delete Feature Flag Decaration
match
@Flag(":[stale_flag]")
:[modifier] boolean :[mthd_name] (){ :[body] }
replace with
using stale_flag is_seed= true

        | Global
        v

Update feature flag check
match :[mthd_name]()
replace with :[value] using value, mthd_name
```

```
class Flags {
- @Flag("location")
- public static boolean isLocEnabled() { ... }
}

class Consumer{
  fun consume() {
- if(isLocEnabled() || x > 0){
+ if(true || x > 0 ){
      return 1
   }
  return 2
 }
}
```

(a) Rule graph (b) Source Code Update

Fig. 9. Rules to cleanup stale feature flag `location` applied to the motivation example in Figure 2

applications). In short, a rule can reference the input substitutions provided in the program, or any capture group that has been recorded when previous rules were applied.

The example in Figure 9 showcases the usage of these `holes`. The first rule in Figure 9b deletes a `public` field that declares the stale flag. At run time, the string `location` will be substituted for `:[stale_flag]` in the rule as given by the input substitutions. Upon matching, this rule will capture the name of the method used to check this flag as `:[mthd_name]`, which is recorded in a symbol table. The next rule `Update feature flag check` uses the captured method name (`:[mthd_name]`) to substitute all of its invocations with `:[value]` (holding the value `true`, from the input substitutions).

3.2.5 Groups. The **`belongs_to`** keyword serves as syntatic sugar to group rules under a common name. Users can reference these named groups in the edge declarations as a shorthand to create an edge between a rule and all rules in a group. For example, creating an edge between a rule and the `boolean_cleanup` group serves as shorthand for linking the rule to every rule in that group.

3.3 Semantics-Driven Design

As authors, we designed the DSL using a *semantics-driven* approach [17]. We divided the domain of code transformation into several sub-domains and crafted a *micro DSL* for each:

- the *tree-sitter queries*, *concrete syntax*, and *regex* handle code matching,
- the *filter* language for refining the matches, and
- the *edge declarations* for defining flow between rules.

Finally, we designed the syntax for `rule` and `program` that integrates these micro-DSLs (Figure 4).

4 LANGUAGE RUNTIME

4.1 Overview

Algorithm 1 provides a high level overview for the language implementation and runtime. The core idea is to maintain a queue of *seed rules*, and traverse the graph and the files in the codebase starting from each seed rule. First, we validate the rule graph to prevent unexpected behavior using a data-flow analysis and syntactic checks on the rules (Line 1). After the validation, we push the seed rules into a global queue and initialize an environment / symbol table with the input substitutions (Line 3 - 4). The environment is used to store both the initial set of substitutions as well as the

Algorithm 1 Core procedure for transforming code given a graph of rules

```
Input:
    (ℛ : RuleGraph, 𝒮 : substitutions, C : path to codebase)
 1: if ¬VALIDATE(ℛ, 𝒮) then
 2:   return
 3: Q ← SEEDRULES(ℛ)
 4: env ← 𝒮
 5: while NOTEMPTY(Q) do
 6:   rule, _ ← POP(Q)
 7:   loop
 8:     isApplied ← false
 9:     for file in RELEVANT(C, rule, env) do
10:       isApplied ∨ =
11:       EXECUTERULEGRAPH(rule, file, ℛ, Q, env)
12:     if ¬isApplied then
13:       break
```

Algorithm 2 EXECUTERULEGRAPH function

```
 1: function EXECUTERULEGRAPH(rule, file, ℛ, mut Q, mut env)
 2:   rulesStack ← [(rule, file)]
 3:   isApplied ← false
 4:   while NOTEMPTY(rulesStack) do
 5:     rule, scope ← POP(rulesStack)
 6:     rule ← INSTANTIATE(rule, env)
 7:     while HASMATCH(rule, scope) do
 8:       match ← GETMATCH(rule, scope)
 9:       isApplied ← true
10:       APPLYEDITS(match, rule, mut env)
11:       for rule, scScope in SUCCESSORS(rule, ℛ) do
12:         if scScope ≡ GLOBAL then
13:           PUSH(Q, (rule, GLOBAL))
14:         else
15:           scope ← RESOLVE(match, file, scScope)
16:           PUSH(rulesStack, (rule, scope))
17:   return isApplied
```

captured groups of from rule executions, which can be used as dynamic elements in subsequent rules. Each seed rule is applied across the entire codebase recursively in a depth-first fashion (Line 5), until no rules match (Line 7 - 13). For each *relevant* file (e.g., a file that is likely to contain the match template of the rule, see Section 4.1.3), we invoke `ExecuteRuleGraph` (Algorithm 2). In this step, the tool traverses over the CSTs and transforms the source code. For each match, it explores the rule graph and stacks the rules in a DFS-manner (Line 11), applying them exhaustively within the scope. The function `ExecuteRuleGraph` is *not pure*, it updates the environment, transforms the source code in-place, and pushes new rules into the queue (Q). We detail each function of the algorithm more thoroughly in subsequent sections.

4.1.1 *Graph Validation.*

The first step in the core algorithm is to verify the graph (Line 1). In our implementation, PolyglotPiranha statically validates the constructed graph to prevent unexpected behavior when the graph is applied to the codebase. First, PolyglotPiranha checks if the individual rules' matchers and filters are well-formed. For example, PolyglotPiranha ensures that each regex compiles and that each s-expression parses correctly according to the language's grammar. It also conducts a data-flow analysis to ensure that no path in the graph traversal leads to a rule where an input variable is not initialized correctly. This is implemented as a definite assignment analysis [23]. If the graph is incorrect, PolyglotPiranha alerts the user to prevent panics that could result from accessing undefined variables.

4.1.2 *Environment.*

The environment is a simple symbol table, which is initialized with the substitutions from the program (Figure 4). Rules can access symbol table variables if they have been declared. If a rule is triggered and a match is found, the symbol table is updated by binding the matched source code to the corresponding named captured group in the symbol table. If a variable already exists in the symbol table, its entry gets over written. Therefore a rule always gets instantiated with the most recent binding of the referenced symbol from the environment. This kind of dynamic variable scoping can also be observed in languages like `LaTeX` or `Bash`.

4.1.3 *Relevancy Check for Performance.*

In rewriting large code bases, repeatedly parsing the entire codebase is inefficient, especially in monorepos with millions of lines. The goal of the function `relevant` is to optimize code rewriting by only parsing files whose content matches the concrete values assigned to the holes of the *global rules* (Line 9). In practice, the concrete values to the input substitutions are used to filter out files that are not relevant to the transformation using string matching. For example, in Figure 9 PolyglotPiranha would only parse the files that contain the string `location`. This is because `stale_flag` is mapped to `location`, and `stale_flag` is a hole in the rule `Delete Feature Flag Declaration`. This simple insight improves PolyglotPiranha's overall performance. The implementation of PolyglotPiranha further boosts this by parallelizing the lookup using fork-join frameworks (like `Comby`). Note that, PolyglotPiranha circumvents this optimization for holes that are referenced inside the `not_contains` or `not_enclosing` clause.

4.2 Rule Graph Execution

Algorithm 2 describes the procedure executeRuleGraph that applies a given *rule* across a *file*. Each time a *seed rule* is triggered, we initialize a *stack* (`ruleStack`) for depth-first traversal of the rule graph (Line 2). Then, we pop rules from stack and apply each rule exhaustively within the specified scope (until `hasMatch` is `false` as shown in Line 7). For each match, we update the environment with the new capture groups and transform the source code by applying the rule (Line 10, and Algorithm 3). Additionally, we also delete any associated commas and comments if necessary (Section 4.2.1). Finally, we add the successors of the current rule in the graph to the local stack or the global queue, and continue this until fix point (Lines 11 - 16).

Algorithm 3 ApplyEdits function

```
1: function ApplyEdits(match, rule, mut env)
2:   updateEnv(match, env)
3:   edit ← getEdit(match, rule)
4:   if isDelete(edit) then
5:     edit ← deleteCommaComments(edit)
6:   apply(edit)
```

4.2.1 Deleting Associated Comments and Trailing Commas. When we first deployed PolyglotPiranha based tools at `Uber`, we quickly realized that just deleting the source code without addressing the comments was not sufficient to get a change approved by code-reviewers. For instance, if we remove the flag `location` in Figure 1, it would also be crucial to delete any associated trailing comment, such as `// Declares Location`, for maintainability purposes. PolyglotPiranha deletes trailing commas, trailing comments, and leading comments, a node is deleted from the parse tree.

It reasons about the immediate sibling (in the parse tree) of the deleted nodes. If the next sibling is a comma, it updates the edit to delete this trailing comma. It then checks if the next or the previous sibling is a comment on the same line (in the source code text) as the deleted node, and includes it in the deletion. Finally, it checks if the previous node is a comment node and if it is the only node that starts at that line in the source code text. If so, it is included in the deletion; and this last step is performed recursively until no such comments are found. In our implementation, the user can provide regex based rules to exclude specific (or all) comments from being cleaned up depending on the programming language at hand.

4.2.2 Optimizations. PolyglotPiranha uses the `tree-sitter` [13] framework for parsing the source code. PolyglotPiranha maintains only one parse tree *object* in its memory, and updates this object sequentially leveraging the `tree-sitter`'s incremental parsing feature. This eliminates the need to parse the file again from scratch after the rewrite, thus optimizing PolyglotPiranha's overall performance. Additionally, to minimize the impact on the parse tree, by default our approach (1) orders the rules from inner to outer scope: starting from the parent, to method, class, file, and finally to global scope, and (2) rewrites code bottom up. These optimizations are effective and eliminate the need for multi-threading.

5 EVALUATION

We seek to answer the following research questions:

RQ1. [Expressiveness] *How expressive is the DSL for real-world code transformation tasks?* We assess this through *three* case studies, focusing on high-impact refactorings crucial to `Uber`'s needs. We highlight the complexity of each, and how to encode it in the DSL.
RQ2. [Effectiveness and Usefulness] *How effective is PolyglotPiranha at automating code changes? To what extent are PolyglotPiranha-based tools useful in practice?* We run the above tools across `Uber`'s proprietary codebase, and measure the percentage of Pull Requests (PRs) [27] that pass Continuous Integration (CI) [20] and are merged without manual intervention. For PRs with intervention, we measure the `LoC` changed by tools versus developer.
RQ3. [Comparison with *state-of-the-art*] *How do PolyglotPiranha-based tools compare to similar tools built upon state-of-the-art frameworks?* We compare the PolyglotPiranha-based implementation against the imperative variants developed upon `ErrorProne` [1] and `OpenRewrite` [41], and against its declarative variants developed upon `Comby` [58] (a lightweight tool). We compare implementations in terms of size, complexity and performance.

5.1 RQ1. Expressiveness

Experimental Setup. To showcase the *expressiveness* of the DSL we present *three* real-world case studies where we automate complex code transformation tasks using PolyglotPiranha. In each case study, we highlight the complexity of the task, and how the DSL can be used to encode it. We chose these three case studies because they are high-impact tasks crucial to `Uber`'s operational needs and they are representative of the tasks that `Uber` or other software companies would want to automate. Moreover, these tasks are not trivial to automate using existing frameworks.

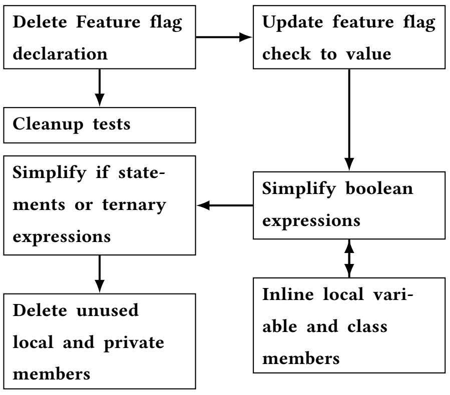

Delete Feature flag declaration
Update feature flag check to value
Cleanup tests
Simplify if statements or ternary expressions
Simplify boolean expressions
Delete unused local and private members
Inline local variable and class members

Fig. 10. Strongly connected components of the rule graph for feature flag cleanup.

5.1.1 Case Study: Stale Feature Flag Cleanup. Feature flagging is a widely adopted and highly encouraged practice at `Uber` [5], and other major software companies [51, 52]. It allows developers to modify configurations without redeploying, supporting A/B testing in production. However, feature flags often become stale, and retaining them beyond their original purpose can lead to technical debt. Therefore, it is important to automate their removal. Indeed, researchers [53] have developed the `Piranha` tool for this purpose. `Piranha` is built on top of the `ErrorProne` [1] frameworks for `Java` and `SwiftSyntax` [3] for `Swift`. However, `Uber`'s codebase uses `Kotlin` and `Go` too. Instead of developing two new language-specific tools, we used PolyglotPiranha to implement this transformation as ***one*** tool supporting `Java`, `Kotlin`, `Swift` and `Go`.

Figure 10 shows the strategy that we implemented for automating the cleanup of stale feature flags at `Uber`. Each node in this figure is a strongly connected component or sub-graph of the original large graph that was applied at `Uber`. Here, each subgraph is a cleanup category. For instance, `Simplify boolean expressions` contains rules that simplify nested boolean expressions with conjunctions, disjunctions and negations. These rules are recursively applied until the expression cannot be

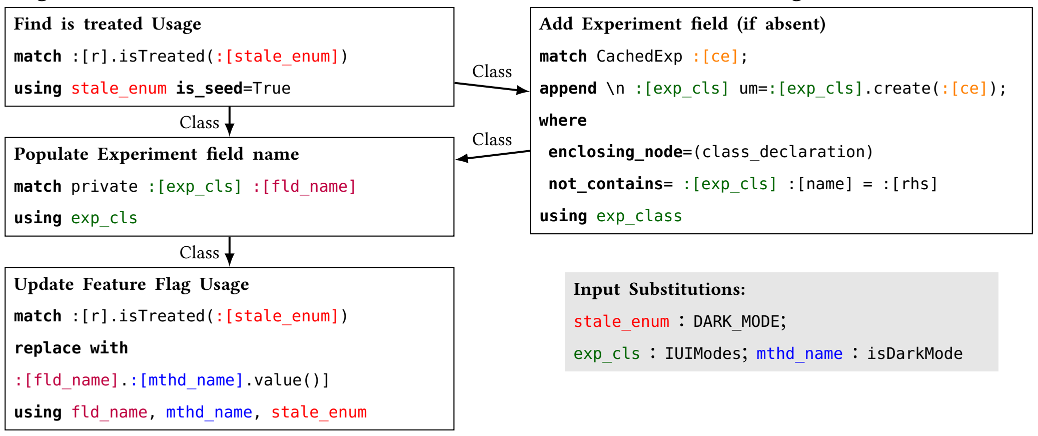
[5]In fact, our motivating example is a simplified version of feature flag cleanup we performed internally.

```
- public enum IUIModesEnum {
+ public interface IUIModes {
   - DARK_MODE,
   + @Param(key="DARK_MODE")
   + BoolParam isDarkMode();
   - LIGHT_MODE,
   + @Param(key="LIGHT_MODE")
   + BoolParam isLightMode(); }
```

(a) Example migration from enum-based feature flag declaration to annotations.

```
class Consumer {
  CachedExp ce = new Experiment();
+ IUIModes um = IUIModes.create(ce);
  public String color() {
- return ce.isTreated(DARK_MODE)
+ return um.isDarkMode().value()
     ? "Black"  : "White";
 }
}
```

(b) Source code update after the migration of enums to interfaces as shown in Figure 11a.



```
Find is treated Usage
match :[r].isTreated(:[stale_enum])
using stale_enum is_seed=True
```

```
Populate Experiment field name
match private :[exp_cls] :[fld_name]
using exp_cls
```

```
Update Feature Flag Usage
match :[r].isTreated(:[stale_enum])
replace with
:[fld_name].:[mthd_name].value()]
using fld_name, mthd_name, stale_enum
```

```
Add Experiment field (if absent)
match CachedExp :[ce];
append \n :[exp_cls] um=:[exp_cls].create(:[ce]);
where
 enclosing_node=(class_declaration)
 not_contains= :[exp_cls] :[name] = :[rhs]
using exp_class
```

```
Input Substitutions:
stale_enum : DARK_MODE;
exp_cls : IUIModes; mthd_name : isDarkMode
```

(c) Part of the original rule graph that migrates usages of the `isTreated` API. The input substitutions in the bottom right instantiates this graph to migrate the `DARK_MODE` feature flag described in this figure.

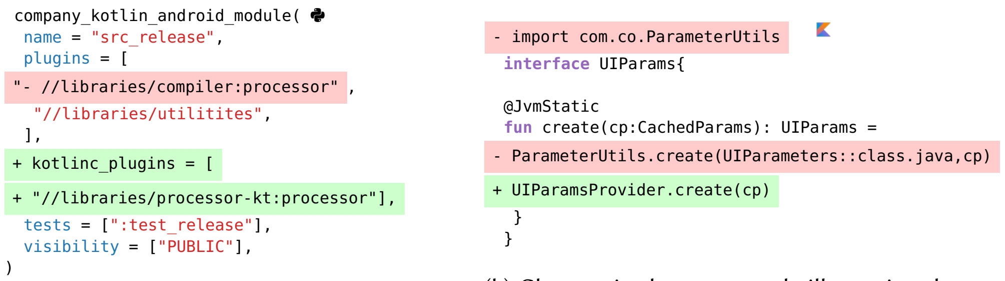
Fig. 11. Experimentation API usage update after the migration from enum-based feature flag declarations.

further simplified. It should be noted how the `Simplify boolean expressions` and `Inline local variables and members` call each other, until no more simplification is possible. The `Cleanup tests` sub-graph is particularly interesting. In this sub-graph we identify all the tests that explicitly set the feature flag to a specific Boolean value. If the set value is the same as the status of the feature flag we elide the setter, else we delete the test case.

Application. We applied this tool across the `Uber`'s Android and iOS codebase. For Android, an additional challenge was the `Java` and `Kotlin` interplay, which PolyglotPiranha handles natively as discussed in Section 4. The *Experimentation* team at `Uber` provides a live list of stale feature flags based on runtime values and various other factors. The tool is deployed at `Uber`, and it continuously generates a PR for any new stale feature flag. Our `Java` and `Swift` implementations are behaviourally equivalent to `Piranha`, which was proposed by previous researchers for the same task. Our tool's output passes the tests from the extensive benchmark scenarios that the authors of `Piranha` maintain, based on their experiences of running `Piranha` at `Uber`.

5.1.2 Case Study: Experimentation API Migration. The *Experimentation* team at `Uber` developed a new *feature flagging* API to support its growing needs. It was imperative for `Uber` to transform thousands of lines of their `Android` code to use this new API.

Figure 11 showcases the code changes required for the migration. The previous *feature-flag* API declared feature flags using `enum` data types. To adapt the code to the new API, these enums need to



```
company_kotlin_android_module(
 name = "src_release",
 plugins = [
"- //libraries/compiler:processor" ,
  "//libraries/utilitites",
 ],
+ kotlinc_plugins = [
+ "//libraries/processor-kt:processor"],
 tests = [":test_release"],
 visibility = ["PUBLIC"],
)
```

(a) Update to the BUCK file

```
- import com.co.ParameterUtils
  interface UIParams{

  @JvmStatic
  fun create(cp:CachedParams): UIParams =
- ParameterUtils.create(UIParameters::class.java,cp)
+ UIParamsProvider.create(cp)
   }
  }
```

(b) Changes in the source code illustrating the usage of the new Kotlin-based processor

Fig. 12. Examples of modifications in the BUCK and Kotlin files for the annotation processor migration.

be rewritten as *annotated abstract methods* (as shown in Figure 11a). These annotations were added to specify metadata information for a feature flag such as *key* and *namespace*. After migrating the `enum` to an interface, this change has to be propagated. For example, consider the feature flag usage in Figure 11b. Previously, the `isTreated` method (Line 5) was invoked to check the status of the feature flag by passing the enum `DARK_MODE`, declared in Figure 11a. However, with the new design clients are expected invoke the feature flag method `isDarkMode()` as shown in Line 6.

In practice, this migration has to accommodate many other *caveats*. To complete this migration, it is necessary to also add new fields (e.g., `IUIModes` (Line 3, Figure 11b). This is handled by writing two rules as shown in Figure 11c : (1) `Add Experiment field` - adds a field of type `IUIMode` (if absent), and (2) `Populate Experiment field name` captures the name of the field of type `IUIMode`. The field name (i.e.: `[fld_name]`) is used in the following rule `Update Feature Flag Usage`, which is the actual rule used to replace the `isTreated` API. Other nuances include deleting consequently unused members and imports and adapting test cases accordingly.

Application. This migration was executed on the `Android` codebase, involving 860 experimentation feature flags using the older API. Feature flags were first grouped based on package and enum declarations. For each flag in the group, the tool was applied and then a PR was created for that group. The *Experimentation* team shepherded these PRs. The rule graph has 28 rules.

5.1.3 Case Study: Annotation Processor Migration. The goal of this migration is to transition the `Android` codebase from a `Java`-based annotation processor to a `Kotlin`-based system to improve overall performance. The changes required for this migration are described in Figure 12.

This migration requires changing all the build configurations (written in BUCK [38]) to be adapted by replacing the old processor dependency with the new one as depicted in Figure 12a. Besides the build files, it is necessary to migrate all `Kotlin` files that initially used the `Java` processor (shown in Figure 12b). For example, Figure 12b shows how `ParameterUtils.create` is replaced with a `Kotlin` equivalent method, `create`, and the unnecessary import statement is deleted.

Application. This migration was orchestrated by a single engineer. The rule graph contained six rules. A PR was created for 25 predefined sub-directories (of the `Android` codebase).

5.2 RQ2. Effectiveness and Usefulness

Experimental Setup. To evaluate the effectiveness of PolyglotPiranha's framework, we evaluate the three tools from the case studies above by applying them to `Uber`'s proprietary codebases. Specifically, the `Android` codebase is composed of 7.5M `LoC` of `Java` and 2.5M `LoC` of `Kotlin`,

Table 1. PRS created and merged by the tool, as well as the % of LOC automatically deleted for each.

		Effectiveness		Usefulness		
Application	**Language**	# PRs	# PRs (CI passes)	# PRs (Accepted)	# files updated	#+/- Lines
Stale Feature Flag Cleanup	Java & Kotlin	2515	1413	817	2952	+15032 /-107635 †
	Swift	2186	1309	614	1733	+ 21230 /-104721 ‡
Experimentation API migration	Java	155	89	155	2146	+19157 /-19041 §
Annotation processor migration	Kotlin & Python	25	25	25	2042	+2809 /-3897 ‖

† 85.7% was automated ‡ 95.3% was automated § 73.4% was automated ‖ 100% was automated

while the iOS codebase is composed of 7.5M LoC of Swift. The PRs produced by our tools are reviewed by the appropriate teams, and merged if they pass the *Continuous Integration* [20] checks and tests. The PRs that fail CI are expected to be manually fixed by the respective team before merging.

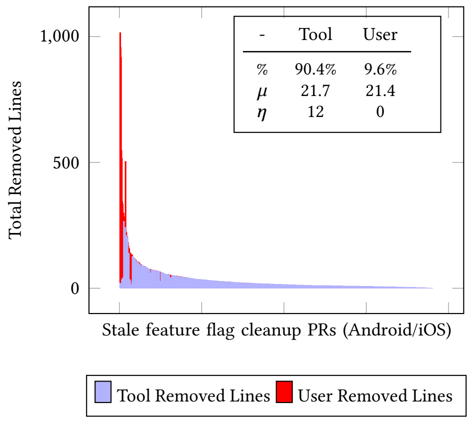
Results Summary. Table 1 summarizes the overall results we obtained by running PolyglotPiranha based tools over our proprietary corpora. For each application, it reports the number of PRs created, PRs accepted (and merged), and PRs that pass the Continuous Integration checks and tests. At large, the three tools produced 4881 PRs in the last six months of which 1611 have been accepted and merged into the main codebase at the time of writing this paper. Particularly, for *stale feature flag cleanup* our acceptance rate is 52.5% (of PRs that pass CI) while for the migrations it is unsurprisingly 100% (because the migrations were orchestrated centrally). These PRs have deleted over 200k LoC of dead code and migrated over 20k LoC of old code to use the new APIs.

5.2.1 *Quality of the Automation.*

Stale Feature Flag Cleanup. The data for this experiment was collected between April and November of 2023. PolyglotPiranha created a total of 4701 PRs, and reviewers did some kind of activity on 1727 (36.7%) of the total number of PRs. These activities include, accepting the PR and merging it, commenting the PR, or patching the PR before accepting it. There are still 1410 PRs that pass all CI checks and are still in queue for review. Further, the reviewers have marked 114 PRs as `Needs Changes` status indicating that the they expect extra cleanup from the tooling. For most of these PRs, the reviewers have reported issues with new features and bugs. The reviewers abandoned 182 PRs, to assert that the cleaned up feature flags are not stale (i.e., the experiment is not over).

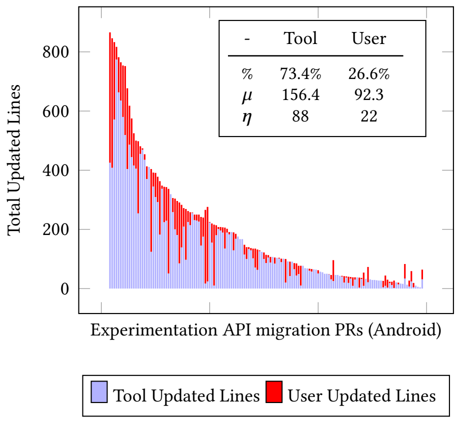
We observed that 56.2% of all the `Android` PRs and and 59.9% of the `iOS` PRs passed all CI checks. The `Uber`'s CI not only builds and tests the change, but it also employs over a hundred linters and bug-checkers to ensure the quality of the change meets the `Uber`'s high standards. These checkers ensure there are no unreachable and unreferenced elements (e.g. `UnusedMethod` check [43]), no sub-optimal code (e.g. `ComplexBooleanConstant` check [44]) and no nullability errors [9].

To further investigate why PRs were failing CI, we sampled 233 PRs from the 1979 failing PRs (confidence level: 95% and margin of error: 6%) for manual perusal. Two of the authors (with over 5 years of research and development experience) coded the failure reasons for the PRs using the thematic analysis guidelines [55]. They first refined the code set on 20% of the PRs. Then, using this code set, they independently coded another 20% of the patterns and came to a high level of agreement. Then they coded the remaining corpus.



-	Tool	User
%	90.4%	9.6%
μ	21.7	21.4
η	12	0

(a) Lines deleted by feature flag cleanup PRs



-	Tool	User
%	73.4%	26.6%
μ	156.4	92.3
η	88	22

(b) Lines updated by API migration PRs

Fig. 13. Lines deleted/updated by tool (blue) vs users (red)

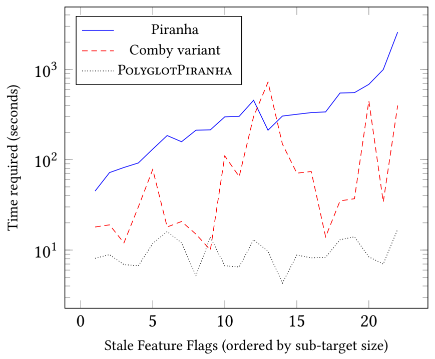
Figure 14 summarizes our observations. We categorized 99 PRs as *Test failures*, where they failed to pass one of the integration or unit test cases. The 72 PRs marked as *Incomplete cleanup* failed one of the linter or static analysis checks. In both categories, the failures stem from the fact that PolyglotPiranha did not complete the cleanup. Most of these failures can be resolved by refining rule graphs to be more comprehensive and account for more edge cases, a process we have iteratively conducted at `Uber`. However, we noticed that some PRs that fail linters cannot be fixed by refining rules due to PolyglotPiranha 's conservative approach in computing def-use chains and call-graphs. This leads to being unable to delete/inline private variables/private-methods in presence of variable shadowing or re-initialization (see Section 6).

In the 28 (12%) PRs in the *unsupported pattern* category, the build target failed to compile because the tool was unable to cleanup all the usages of the feature flag. These failures are expected and happen by design. The *experimentation API* maintains a strict coding guideline for its usage, and the maintainers decided not to provide any automated cleanups/migration support when the convention is violated. The failing 23 PRs (9.9%) in the *over deletion* category failed to compile because PolyglotPiranha wrongly deleted code due to unsound/incorrect rules. A large portion of these correspond to the iOS variant, that currently does not reason about the usages of private methods in *struct implementations* in other files. Therefore, it ends up unnecessarily deleting private methods. We also identified bugs in the implementation in 7 (3%) PRs (*bugs in the tool*), which we will fix in future releases. Additionally, there was a problem with the bot that orchestrates PolyglotPiranha PRs internally in 4 (1.7%) PRs (*infra issues*).

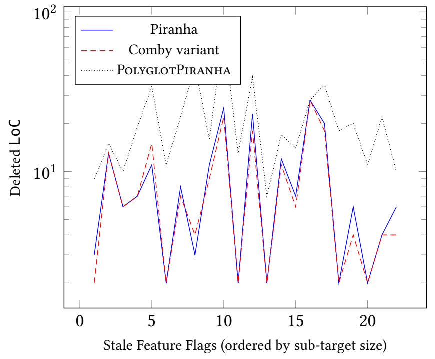
Category	**Frequency**
Test Failure	99
Incomplete Cleanup	72
Unsupported Patterns	28
Over Deletion	23
Bugs in the tool	7
Infra issues	4

Fig. 14. Analysis of 233 PRs failing CI

Experimentation API. For the *Experimentation API migration*, we observed that 89 (59.9%) PRs passed all CI checks. The main reason migration PRs to fail was non-standardized usage of the API and usage of some specific API patterns that were not automated. Nonetheless, the tool still automated 73.4% of all lines deleted. The migration was driven centrally by the team, therefore



(a) Time required to perform the cleanup. Flags are ordered by target size.



(b) Lines of code deleted for each tool. Flags are ordered by target size.

Fig. 15. Comparative analysis of `Comby`, `Piranha`, and `PolyglotPiranha` for stale feature flag clean up.

the all the PRs were immediately acted upon after creation. The team reviewed these PRs, patched them if necessary and landed them.

5.2.2 *Automation Ability.* To study the manual effort involved in each merged PR, we compute the number of lines removed by the tooling automatically and the subsequent manual effort. Figure 13 shows this data for for each *stale feature flag cleanup* and *Experimentation API migration* PR that was merged, as a stacked bar chart. The *annotation processor* migration was fully automated.

Stale Feature Flag Cleanup. The tooling has deleted 85.7% and 95.3% of all the total deleted lines across the `Android` and `iOS` codebases (90.4% gross) respectively across all merged PRs, as shown in Figure 13a. We observed that 75.9% of the PRs that were merged required no user intervention. However when the developer did intervene, they deleted a lot of code before merging the PR, hence the mean number of lines deleted by user is skewed ($\mu = 21.4$, $\eta = 0$). In few outlier cases developer deleted more than 900 lines of code. Probing further into these outlier PRs, we discovered that developers had removed a collection of top-level classes that were guarded by the flag. Some of these scenarios will be incorporated into the next version of our tool. However, very precise and general support for such cleanups is impractical in our lightweight approach.

Experimentation API Migration. The tooling has migrated 73.4% of the total lines deleted, however we observed that more than 74.8% PRs needed some manual intervention. In these cases developers on average updated another 92 lines upon the changes proposed by the tool. We observed that `Uber` developers also made manual changes to the PRs that pass CI. These changes include class deletions, removing unused data files, updating comments and method names. While refining rules can resolve certain scenarios, some require symbol or type information, and others, such as method renaming and updating documentation, are beyond the scope of traditional tools. We also observed that the team knowingly used the tool to perform partial migrations even for cases where all APIs were not supported. The small spikes towards the tail end of the chart show these scenarios.

5.2.3 *Complexity of Changes.* To understand the complexity of the changes, we reason about the number of files touched per change. We observed that an average of 3.63 files ($\eta = 3$) were touched per *stale feature flag cleanup* PR. In six outlier cases 40-50 files were touched, where 90% of them were deleted (there are three cases that delete between 900-1000 `LoC` in Figure 13a). Further investigation revealed that a majority of these files contained supporting classes that were guarded by feature flag cleanups and resource files referenced in these deleted files. More powerful static analysis could potentially find these cases, however our syntactic approach cannot handle general unreachable code.

5.3 RQ3. Comparison with State-of-the-Art Code Rewrite Frameworks

5.3.1 *Performance.*

Experimental Setup. We compare PolyglotPiranha-based stale feature flag cleanup against `Piranha` [53] and an equivalent we develop based upon `Comby` [58]. The `Comby` implementation has 29 rewrite rules for `Java`. It was particularly easy to develop the `Comby` variant because PolyglotPiranha's concrete syntax DSL is inspired by `Comby`. For this evaluation, we chose 24 stale feature flags randomly from the PRs that (1) passed CI but were not accepted (at the time of writing this paper) (2) were used in `Java` files (because `Piranha` only supports `Java`). Note that we only chose 24 feature flags because it takes significant manual effort to integrate `Piranha` within our infrastructure due to `Piranha` depending on compilation[6]. For each feature flag, we noted the affected sub-targets and their sizes. We then applied the three tools across the sub-targets and the execution time was recorded. These experiments were performed on an enterprise-class VM in Google Cloud Platform. Note that we neither compare the quality of the cleanups nor precision because by construction `Comby` uses a more loose representation of code, based on Dyck-extended grammars [58] (i.e., balanced parenthesis grammars), whereas PolyglotPiranha uses language-specific grammars from the `tree-sitter` reportoire, hence PolyglotPiranha transformations are more powerful and precise. Conversely, `ErrorProne` and `OpenRewrite` can leverage semantic information like symbol/name resolution, for higher precision and applicability, but are not polyglot.

Results. The line chart in Figure 15a shows the performance of each of the tools for the set of flags we identified above (ordered by the size of the corresponding *sub-targets*, ranging from 1.2K to 1.8M `LoC`). PolyglotPiranha took an average of 9.74 ± 3.46 seconds, `Comby` 121.67 ± 179.03 seconds (12.32×), and `Piranha` 413.91 ± 521.94 seconds (42.5×). We can see `Piranha`'s execution increases almost linearly with the target size (due to the fact that `Piranha` relies on building the target). PolyglotPiranha and `Comby` depended on the number of passes and files affected for the refactoring. The fact that PolyglotPiranha is faster than the `Comby`-based variant is surprising because `Comby` has a string based matching approach with minimal overhead. These results can be attributed to the fact that `Comby` has no sense of ordering between rules (nor scope), therefore, the match-replace rules are applied across the entire subtarget. PolyglotPiranha's performance is also attributed to optimizations discussed in Section 4.1.3 and 4.2.2.

Figure 15b shows the number of lines deleted by each of the tools for the same set of flags (in the same order). PolyglotPiranha deletes more lines of code because it's able to delete trailing commas and comments. Note that we manually vetted that there are no over deletions in these PRs. In summary, PolyglotPiranha is consistently faster while deleting more lines than its imperative alternative `Piranha` and a lightweight `Comby`-based alternative.

5.3.2 *Expressiveness and Ease-of-Use.*

Experimental Setup. We compare the implementation of PolyglotPiranha-based tools against their imperative variants. Specifically, we compare the PolyglotPiranha-based Stale Feature flag cleanup program against the implementation of `Piranha` [53]. Further we also encode three *pre-existing* code transformation recipes developed by *professional tool builders*, specifically `OpenRewrite` - (1) (`JHipster Upgrade`) Fix CWE-338 with SecureRandom [45] (2) (`Slf4j`) Loggers should be named

[6]While Piranha was developed and was previously integrated at Uber, however, both Uber's feature flag API and developer infrastructure have changed since then

Table 2. Comparison of PolyglotPiranha tools against existing implementations

		Bug Fixes			Feature Flag Cleanup	
Tool	**Metric**	CWE-338	slf4J	java.security	Android	iOS
PolyglotPiranha	LoC	**68**	**31**	**23**	**654**	**1156**
	# rules	4	3	3	31	42
Error-Prone	LoC	-	-	-	3467	-
SwiftSyntax	LoC	-	-	-	-	1316
OpenRewrite	LoC	145	87	92	-	-
Comby	# rules	-	-	-	29[†]	-

[†] This feature flag cleanup variant was developed for the experiments.

for their enclosing classes [46] (3) (`java.security`) Use secure temporary file creation [48]. The selected patterns are (1) related to a popular `Java` library (2) involve multiple interdependent changes (3) have associated test cases for validation (4) clearly fix a bug or security vulnerability.

Stale Feature Flag Cleanup. As discussed in Section 5.3.1, `Piranha` is a stale feature flag cleanup tool with multiple implementations, one for each language supported. This is because `Piranha` is built upon language-specific imperative frameworks for code analysis and rewriting. To compare the expressiveness and conciseness of both approaches, we qualitatively and quantitatively compare the `PiranhaJava` and `PiranhaSwift` variants against their PolyglotPiranha-based counterparts.

`PiranhaJava` is built upon the `ErrorProne` [1] framework, whereas `PiranhaSwift` uses `SwiftSyntax` [3]. Table 2 (right) shows that PolyglotPiranha based approach is significantly more concise in terms of `LoC`. Moreover, rules can be re-used across languages (e.g., simplify disjunction in Figure 3 is language-agnostic). PolyglotPiranha-based `Swift` variant is more powerful than `PiranhaSwift` (e.g., supports variable inlining and cleanup of the unused members).

In contrast, our `Comby` implementation for feature flag cleanup in Java comprises 29 rules. Due to `Comby`'s limitations, we were unable to express 10 transformations from our PolyglotPiranha implementation, including inlining singly-used boolean variables, deleting unused fields and variables, removing unnecessarly nested blocks, and deleting files under certain conditions and enum blocks. Despite this, the rule count difference is minor: 31 for PolyglotPiranha versus 29 for `Comby`. This is because PolyglotPiranha allows for the use of different, more powerful transformation languages. For instance, `tree-sitter` queries provide a syntax for complex alternations. Therefore, the `Comby` variant ends up being more verbose, requiring additional rules for the same task.

OpenRewrite. `OpenRewrite` project is a semantic code search and transformation ecosystem. Its platform allows writing code transformation recipes for common framework migration and stylistic consistency tasks. We picked three relevant recipes written by professional developers, corresponding to high-impact transformations. We implemented the same refactoring actions using PolyglotPiranha. Table 2 shows the `LoC` count and number of rules for both PolyglotPiranha and `OpenRewrite` recipes. Our implementations pass the tests of the `OpenRewrite` recipes.

6 LIMITATIONS AND DISCUSSION

Transformation Correctness. PolyglotPiranha does not guarantee that the transformed code will compile, be semantically correct, or precisely reflect the developer's intent. This limitation is common to other syntax-driven code transformation tools such as [5, 18, 58]. While our dataflow analysis verifies the rule graph's consistency and grammatical accuracy (Section 4.1.1), the effectiveness and accuracy of transformations ultimately rely on the quality of the rule graph itself.

Syntactic Limitations. PolyglotPiranha's purely syntactic approach limits its ability to perform transformations that require semantic information of the code. In practice, this means that code rewrites that require type resolution, class hierarchy analysis, and/or control-flow analysis can not be expressed in the DSL today. Specifically, PolyglotPiranha: (1) *lacks precise def-use information.* We designed rules conservatively to identify def-use relationships within the syntactic scope of the variable declaration. However, due to the lack of SSA representation and dominator information PolyglotPiranha cannot reason about variable shadowing or re-initialization. (2) *lacks precise type information.* We approximate type information by analyzing declarations within a scope. This falls short when dealing with language features that obscure type information, such as `Java`'s `var` keyword or dynamically typed languages like `Python`. (3) *lacks call-graph analysis.* We approximate caller-callee relationships using method names and their number of arguments, resulting in imprecision in the presence of interfaces, class hierarchies, and method overloading. (4) *cannot handle advanced language features* that require semantic analysis, such as reflection.

Despite these limitations, our evaluation showcases that PolyglotPiranha is effective at automating three real-world code transformation tasks. Though imperfect, even in cases where it was partial, this automation substantially alleviated developers' load as seen in Figure 13a.

Supporting New Languages. At `Uber`, PolyglotPiranha supports languages beyond the ones listed in the evaluation, including `Go`, `Python`, `Scala`, `Typescript`, as well as protocol formats like `Thrift`. PolyglotPiranha uses `tree-sitter` for code parsing, thus supporting a new language requires: (1) incorporating the `tree-sitter` grammar within PolyglotPiranha, and (2) authoring scope-capturing rules in a configuration file (i.e., one rule per scope such as class, method, or file). PolyglotPiranha uses these scopes when applying rules from the rule graph. Note that `tree-sitter` officially supports 133 programming languages [13], including functional languages like `Haskell` and `Scheme`. In fact, we support `Scheme` as a language in PolyglotPiranha, and use it within PolyglotPiranha's implementation for rewriting its structural queries (a subset of `Scheme`). The implementation burden for this support was minimal and comparable to other languages.

Adapting PolyglotPiranha-based tools, like those for feature flag cleanup, to new languages may require additional work. For example, a rule for simplifying a disjunction `(true || :[a])` in `Java` needs to be customized for `Python` as `true or :[a]`. However, we observed that some rules are reusable within a broad family of languages (`Java`, `Kotlin`, etc).

PolyglotPiranha's Usability. To assist users in debugging and root-causing failures due to errors in the rule graph, PolyglotPiranha outputs detailed reports of all executed rules (in order) including their corresponding matched `LoC` ranges, and runtime arguments in an easily queryable format. This allows for step-by-step replay and analysis. Our repository contains examples that explain how to enable debugging mode. We have also developed a playground for rule experimentation that allows users to easily experiment with rules and rules graphs on code snippets. This playground is publicly available on our artifact.

Note that we did not conduct any usability study as part of this paper. However, it's noteworthy to mention that the rules for Feature Flag Cleanup (iOS) and Annotation Processor migration were crafted by iOS and Android app developers at `Uber` who were not familiar with PolyglotPiranha's internals and were not part of our research team.

7 FUTURE WORK

One way to improve precision and circumvent some problems described in Section 6 would be to extend the syntax to support matching semantic information based on type resolution or hierarchy analysis, similar to how previous researchers [31] extended the `Comby` syntax. An option would be to use a Language Server Protocol [39] (LSP) to provide semantic information on variables and bindings. For example, it would be possible to get type information for holes in rules (e.g., `isFlagEnabled(:[x])`, where `:[x]` is of type `string`). However, LSP integration is non-trivial. Since our framework rewrites code in-place, it effectively invalidates some of the information provided by LSP before the rewrite. A solution would be to incorporate incremental analysis frameworks [4]. As an alternative, adding support for `CodeQL` might offer deeper analysis of code over simple `tree-sitter` queries. However, integrating CodeQL involves interfacing with the compiler and build infrastructure, and such effort may not be trivial.

PolyglotPiranha's rules support multiple match-replace languages, catering to different users. It has been observed by [58] that maintaining a tool rooted in declarative code rewrite methodologies is more straightforward than other methods. In Section 5, we demonstrate more concise programs in our DSL compared to their imperative counterparts. With generative AI, learning and assisting users at writing PolyglotPiranha rules is a promising direction to explore.

8 RELATED WORK

8.1 Declarative Code Transformation Tools

Multiple language-specific declarative transformation tool sets, such as `Coccinelle` [35] for C, and `Refaster` [59] for Java, have been proposed. Variants of these tools include `Coccinelle4J` [28] for Java and `GoPatch` [26] for Go. Since they are built using language-specific infrastructure they can leverage semantic information, such as control flow, to enable precise transformations. However, significant efforts are required to introduce and maintain new language front-ends for these tools. In contrast, lightweight tools tools like `Comby` [58] and `ast-grep` [5] provide alternatives that do not rely on a specific compiler infrastructure and are language-agnostic. However lightweight tools generally have limited understanding of code context and semantics, making it challenging to express non-atomic transformations. In contrast, our DSL extends lightweight tools by providing primitives and operations to increase expressiveness and applicability of match-replace rules.

Language workbenches, like `Spoofax` [29] (which incorporates `Stratego` [11]) and `Rascal` [33], offer comprehensive toolsets for designing and implementing languages, including metalanguages for writing code transformations. Similary, `DMS` [10] also provides a declarative rewrite language for transforming code and allows users to combine it with procedural rewrites. However, these tools require the specification of the target language's grammar and its extension to support meta-syntax using the toolset. Unlike language workbenches, PolyglotPiranha does not concern itself with the matching language or the target language. It effectively delegates the code rewrite to a match-replace tool, following a more pragmatic approach. Moreover, PolyglotPiranha introduces rewrite strategies in a new graph-based paradigm that uses minimal meta-syntax.

`TXL` [15] is a multi-language transformation tool that requires users to write both grammar specifications and transformation rules within the `TXL` language. `TXL` transformations often use non-terminal symbols in rewrite rules, which makes them non-trivial to write and use. `Cubix` [34] provides an alternative multi-language solution using compositional data types. `Cubix`, as described by its authors, is not for the lay programmer [34] and requires significant effort and expertise to learn. PolyglotPiranha, on the other hand, offers multiple matching languages ranging for simple regex to structural query languages.

Query languages for code that support complex analyses, namely `CodeQL` [24], could also be used in our rule language, as an alternative to concrete syntax or tree sitter queries. This will enhance precision of rule matching with semantic awareness, and could be useful in some scenarios. Note, however, that `CodeQL` has limited language support, and might introduce performance overheads and require additional integration efforts due to its build system dependency.

8.2 Imperative Code Transformation Tools

Previously researchers and tool builders have invested heavily in the development of advanced refactoring [19, 21, 30, 56], migrations [8, 47, 50] and cleanup tools [53] upon imperative frameworks like such as `Clang LibTooling` [14] for `C/C++`, `ErrorProne` [1] or `OpenRewrite` [41]. In particular, `OpenRewrite` (which is language-specific) offers a framework akin to PolyglotPiranha for crafting reusable recipes. Unlike PolyglotPiranha, `OpenRewrite` uses an imperative approach based on visitors to implement the recipes. While PolyglotPiranha 's rule-graphs can generally be implemented as visitors, the reverse is not necessarily true. Our comparison (Section 5.3.2) suggests that some realistic `OpenRewrite` recipes can be translated into concise rule graphs. Imperative frameworks are usually built around compiler infrastructure and can leverage symbolic information (e.g., name resolution), and other semantic information. Their usage is justified in cases where the some in-depth analysis is needed. However not all code transformations in the real world require such heavy-weight infrastructure (as shown in Section 5.1). On the other hand, our approach provides a declarative DSL to express code transformations as a graph of match-replace rules rather than visitor-style programs. Our implementation leverages the `tree-sitter` framework for parsing the code in multiple languages (necessary for rule application with scopes).

8.3 Program Synthesis

Instead of expecting the user to express the code transformations in a DSL, researchers have proposed *refactoring-by-example* approaches. These approaches infer the transformations as a program in a low level DSL from input-output examples. For example, `LASE` [37] and `Bluepencil` [40] infer an edit script from two example edits. More recently, `Overwatch` [61] integrates refactoring by example ideas into core IDE infrastructure to learn edit sequences, not just from input-output examples but also intermediate steps (i.e. using temporal context). `MELT` [54] and `TC-Infer` [31] uses input-output examples of migrations and type changes from the code history to learn a set of edit rules in the `Comby` language. Potentially, these and other *refactoring-by-example* approaches could learn the transformation as programs in our DSL.

On the other end, black-box synthesis approaches aim to transform code directly rather than generating edit scripts. `SOAR` [42] uses documentation from two closely aligned libraries to migrate code from one to the other, without relying on particular examples and training data. More recently, LLMs like `GPT-4` [49] have been used to identify and automate some non-trivial localized edits from language and examples. However, we are not aware of any comparable LLM-based tools that support the automations described in Section 5.1 at an industry scale.

9 CONCLUSION

Automating code transformations is crucial for increasing maintainability and code quality metrics in growing codebases. To ease this effort, developers rely on automated code transformation languages and toolsets. We propose a new DSL, as well as a tool based on the language named PolyglotPiranha. Our language leverages existing lightweight match-replace paradigms and makes them more generally applicable by providing a set of primitives for cascading and composing rules. The key idea is to use multiple syntactic checks for precise transformations. The approach results in a lightweight, familiar language for automating large-scale changes. We demonstrate the effectiveness of tools developed upon PolyglotPiranha across three use cases, and evaluate them in our proprietary corpora. So far, we have deleted over 210K `LOC` and migrated 20K `LOC` with PolyglotPiranha-based tools. In all use cases, the tool was able to automate between 73.4% and 100% of the necessary changes, yielding significant productivity gains.

ACKNOWLEDGMENTS

We appreciate Satyam Shubham, Adarsh Pandey, and Diego Marcilio for being the initial users of the PolyglotPiranha DSL and offering valuable feedback. We would also like to thank Nada Attia for developing the infrastructure to automatically create diffs. Additionally, we extend our thanks to Arun Babu ASP and Mahesh Hada from the Experimentation team, as well as the Android and iOS monorepo teams, for their support to operationalize PolyglotPiranha at Uber.

Daniel Ramos is supported by Portuguese national funds through FCT, Fundação para a Ciência e a Tecnologia, under PhD grant SFRH/BD/150688/2020. All statements made are those of the authors and do not necessarily reflect the views of any funding agency.

REFERENCES

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *Proc. International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, Riva del Garda, Italy, 14–23. https://doi.org/10.1109/SCAM.2012.28

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, USA. https://www.worldcat.org/oclc/12285707

[3] Apple. 2024. *SwiftSyntax Documentation*. Apple Inc. https://swiftpackageindex.com/apple/swift-syntax/509.0.0/documentation/swiftsyntax Accessed: 2024-03-06.

[4] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 288–298. https://doi.org/10.1145/2568225.2568243

[5] Astgrep. 2024. *ast-grep: write code to match code*. Open-source. https://ast-grep.github.io Accessed: 2024-03-06.

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

[7] Sora Bae, Sungho Lee, and Sukyoung Ryu. 2019. Towards understanding and reasoning about Android interoperations. In *Proc. International Conference on Software Engineering (ICSE)*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, Montreal, QC, Canada, 223–233.

[8] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 265–279. https://doi.org/10.1145/1094811.1094832

[9] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: practical type-based null safety for Java. In *Proc. Symposium on the Foundations of Software Engineering (FSE)*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, Tallinn, Estonia, 740–750. https://doi.org/10.1145/3338906.3338919

[10] Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. 2004. DMS®: Program Transformations for Practical Scalable Software Evolution. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 625–634. https://doi.org/10.1109/ICSE.2004.1317484

[11] 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.

[12] Max Brunsfeld. 2018. Tree-sitter: A New Parsing System for Programming Tools. Strange Loop Conference. Available online: https://thestrangeloop.com/2018/tree-sitter---a-new-parsing-system-for-programming-tools.html.

[13] Max Brunsfeld. 2024. *Tree-sitter*. GitHub. https://tree-sitter.github.io/tree-sitter/ Accessed: 2024-03-06.

[14] Clang. 2024. *Clang LibTooling*. Accessed: 2024-03-06.

[15] James R Cordy. 2004. TXL-a language for programming language tools and applications. *Electronic notes in theoretical computer science* 110 (2004), 3–31.

[16] Detekt. 2024. Detekt: A static code analyzer for Kotlin. https://detekt.dev Accessed: 2024-03-06.

[17] Martin Erwig and Eric Walkingshaw. 2012. Semantics-driven DSL design. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments* 1 (01 2012), 56–80. https://doi.org/10.4018/978-1-4666-2092-6.ch003

[18] Fastmod. 2024. *fastmod*. Meta Inc. https://github.com/facebookincubator/fastmod Accessed: 2024-03-06.

[19] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: Identification and Application of Extract Class Refactorings. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 1037–1039. https://doi.org/10.1145/1985793.1985989

[20] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[21] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LambdaFicator: From imperative to functional programming through automated refactoring. In *2013 35th International Conference on Software Engineering (ICSE)*. 1287–1290. https://doi.org/10.1109/ICSE.2013.6606699
[22] Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions* (3 ed.). O'Reilly Media, Sebastopol, CA, USA.
[23] Nicu G. Fruja. 2004. The Correctness of the Definite Assignment Analysis in C#. *J. Object Technol.* 3, 9 (2004), 29–52. https://doi.org/10.5381/JOT.2004.3.9.A2
[24] GitHub. 2024. *CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub Advanced Security.* GitHub, Inc. https://codeql.github.com
[25] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.
[26] gopatch. 2024. *go-patch: Structured code diffs and refactors.* Uber Technologies, Inc. https://github.com/uber-go/gopatch GitHub repository.
[27] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proc. International Conference on Software Engineering (ICSE)*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, Hyderabad, India, 345–355. https://doi.org/10.1145/2568225.2568260
[28] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. 2019. Semantic Patches for Java Program Transformation (Experience Report). In *Proc. European Conference on Object-Oriented Programming, ECOOP (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, London, United Kingdom, 22:1–22:27. https://doi.org/10.4230/LIPICS.ECOOP.2019.22
[29] Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proc. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, USA, 444–463. https://doi.org/10.1145/1869459.1869497
[30] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type migration in ultra-large-scale codebases. In *Proc. International Conference on Software Engineering (ICSE)*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, Montreal, QC, Canada, 1142–1153. https://doi.org/10.1109/ICSE.2019.00117
[31] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *44th International Conference on Software Engineering (ICSE '22)* (Pittsburgh, United States) *(ICSE '22)*. ACM. https://doi.org/10.1145/3510003.3510115
[32] Jongwook Kim, Don S. Batory, and Danny Dig. 2015. Scripting parametric refactorings in Java to retrofit design patterns. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, Bremen, Germany, 211–220. https://doi.org/10.1109/ICSM.2015.7332467
[33] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, Edmonton, Alberta, Canada, 168–177. https://doi.org/10.1109/SCAM.2009.28
[34] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One tool, many languages: language-parametric transformation with incremental parametric syntax. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.
[35] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *USENIX Annual Technical Conference*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, Boston, MA, USA, 601–614. https://www.usenix.org/conference/atc18/presentation/lawall
[36] Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87. https://doi.org/10.1145/2854146
[37] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *Proc. International Conference on Software Engineering (ICSE)*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, San Francisco, CA, USA, 502–511. https://doi.org/10.1109/ICSE.2013.6606596
[38] Meta. 2024. Build faster with Buck2: Our open source build system. (April 2024). https://engineering.fb.com/2023/04/06/open-source/buck2-open-source-large-scale-build-system/ Accessed: 2023-11-14.
[39] Microsoft. 2024. *Language Server Protocol.* https://microsoft.github.io/language-server-protocol/
[40] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (oct 2019), 29 pages. https://doi.org/10.1145/3360569
[41] Moderne. 2024. *OpenRewrite: Semantic code search and transformation tool.* Moderne Inc. https://docs.openrewrite.org Accessed: 2024-03-06.
[42] Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. In *Proc. International Conference on Software*

Engineering (ICSE). IEEE, Madrid, Spain, 112–124.

[43] Online. 2024. *ErrorProne ComplexBooleanConstant.* Available: https://errorprone.info/bugpattern/UnusedMethod.

[44] Online. 2024. *ErrorProne ComplexBooleanConstant.* Available: https://errorprone.info/bugpattern/ComplexBooleanConstant.

[45] Online. 2024. *Fix CWE-338 with SecureRandom.* Available: https://docs.openrewrite.org/recipes/java/security/fixcwe338.

[46] Online. 2024. *Loggers should be named for their enclosing classes.* Available: https://docs.openrewrite.org/recipes/java/logging/slf4j/loggersnamedforenclosingclass.

[47] Online. 2024. *Open Rewrite Recipes.* Available: https://docs.openrewrite.org/concepts-explanations/recipes.

[48] Online. 2024. *Use secure temporary file creation.* Available: https://docs.openrewrite.org/recipes/java/security/securetempfilecreation.

[49] OpenAI. 2024. *GPT-4 Technical Report.* OpenAI Inc. Available online: https://ar5iv.org/abs/2303.08774 [Accessed: 2024-03-06.

[50] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. 2022. Towards Automated Library Migrations with Error Prone and Refaster *(SAC '22).* Association for Computing Machinery, New York, NY, USA, 1598–1606. https://doi.org/10.1145/3477314.3507153

[51] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature toggles: practitioner practices and a case study. In *Proc. International Conference on Mining Software Repositories (MSR)*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, Austin, TX, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[52] Md Tajmilur Rahman, Peter C. Rigby, and Emad Shihab. 2019. The modular and feature toggle architectures of Google Chrome. *Springer Empirical Software Engineering (ESE)* 24, 2 (2019), 826–853. https://doi.org/10.1007/S10664-018-9639-0

[53] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha: Reducing feature flag debt at Uber. In *Proc. International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, Seoul, South Korea, 221–230.

[54] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. *Proc. International Conference on Automated Software Engineering (ASE)* (2023). https://doi.org/10.48550/arXiv.2308.14687

[55] J Saldana. 2009. *The coding manual for qualitative researchers.* https://doi.org/10.1108/QROM-08-2016-1408

[56] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, and Syed Ahmed. 2018. Towards Safe Refactoring for Intelligent Parallelization of Java 8 Streams. In *International Conference on Software Engineering: Companion Proceedings* (Gothenburg, Sweden) *(ICSE '18).* ACM/IEEE, ACM, New York, NY, USA, 206–207. https://doi.org/10.1145/3183440.3195098

[57] Uber. 2024. PolyglotPiranha: A flexible multilingual framework for chaining interdependent structural search/replace rules. https://github.com/uber/piranha. Accessed: 2024-03-28.

[58] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, Phoenix, AZ, USA, 363–378.

[59] Louis Wasserman. 2013. Scalable, example-based refactorings with refaster. In *Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013*, Emerson R. Murphy-Hill and Max Schäfer (Eds.). ACM, Indianapolis, IN, USA, 25–28. https://doi.org/10.1145/2541348.2541355

[60] Titus Winters, Tom Manshreck, and Hyrum Wright. 2020. *Software engineering at google: Lessons learned from programming over time.* O'Reilly Media, Sebastopol, CA, USA.

[61] Yuhao Zhang, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, and Ashish Tiwari. 2022. Overwatch: learning patterns in code edit sequences. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 395–423. https://doi.org/10.1145/3563302

Received 2023-11-16; accepted 2024-03-31