Automatic Inference of Translation Rules for Native Cross-Platform Mobile Applications

Kijin An, Na Meng, and Eli Tilevich
Dept. of Computer Science
Virginia Tech
{ankijin,nm8247,tilevich}@cs.vt.edu

ABSTRACT

A native cross-platform mobile app has multiple platform-specific implementations. Typically, an app is developed for one platform and then ported to the remaining ones. Translating an app from one language (e.g., Java) to another (e.g., Swift) by hand is tedious and error-prone, while automated translators either require manually defined translation rules or focus on translating APIs. To automate the translation of native cross-platform apps, we present tInferer, a novel approach that iteratively infers syntactic transformation rules and API mappings from Java to Swift. Given a software corpus in both languages, tInferer first identifies the semantically equivalent code based on braces and string similarity. For each pair of similar code segments, tInferer then creates syntax trees of both languages, leveraging *the minimalist domain knowledge of language correspondence (e.g., operators and markers)* to iteratively align syntax tree nodes, and to infer both syntax and API mapping rules. tInferer represents inferred rules as string templates, stored in a database, to translate code from Java to Swift. We evaluated tInferer with four applications, using one part of the data to infer translation rules, and the other part to apply the rules. With 76% in-project accuracy and 65% cross-project accuracy, tInferer outperforms in accuracy j2swift, a state-of-the-art Java-to-Swift conversion tool. As native cross-platform mobile apps grow in popularity, tInferer can shorten their time to market by automating the tedious and error prone task of source-to-source translation.

ACM Reference format:
Kijin An, Na Meng, and Eli Tilevich. 2016. Automatic Inference of Translation Rules for Native Cross-Platform Mobile Applications. In *Proceedings of ACM MOBILESoft'18, Gothenburg, Sweden, May 2018 (MOBILESoft'18),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To increase market share, software companies and open source organizations release different versions of their mobile apps for multiple mobile platforms. To ensure a satisfactory user experience, mobile developers find themselves having to produce platform-specific implementations of their apps. We refer to such applications as *native cross-platform mobile apps.* As a specific example, a native cross-platform mobile app can have three different versions: one implemented in Java for Android, another implemented in Swift for iOS, and yet another one implemented in C# or C++ for Windows Phone. When developing multi-platform apps, programmers commonly first focus on one platform. Once the app developed for this platform matures, it is then ported to the remaining platforms. For example, first an app can be developed in Java for Android. The resulting Android Java version of the app (i.e., the source) is then translated into an iOS Swift version (i.e., the target). As the source and target programming languages follow different grammars and feature dissimilar software libraries, the developers of native mobile apps must be equally versed in both languages and their APIs to correctly port code. Translating the code of an app by hand can be quite tedious and error-prone, thus motivating the need for approaches that can automate the translation process.

1.1 Related Work

Existing code migration tools require users to manually define the transformation rules [10, 16–18, 24, 31, 32]. However, defining these rules by hand is still laborious and error-prone, as a variety of API and syntax mapping rules need to be specified. For example, Java2CSharp [18] and j2swift [10] can convert program structures based on predefined translation rules, but are unable to translate many APIs due to the large volume of libraries available for different languages.

Zhong et al. [34] and Nguyen et al. [27] automatically mined API usage mappings between Java and C#. Specifically, Zhong et al. aligned the code in two versions based on similar names of classes and methods, and then constructed the API transformation graphs for each pair of aligned statements to identify API mappings [34]. Nguyen et al. mined API usage sequence mappings by conducting program dependency analysis [25] and representing API usage as groums [28]. However, neither approach automatically applies the inferred rules to translate code. mppSMT [26] is a state-of-the-art approach that automatically migrates Java code to C# using phrase-based statistical machine translation. It infers and applies both structure and API mapping rules, as guided by the following two developer-provided kinds of domain knowledge: (1) the basic mapping rules between statement types across languages, such as `SuperCall` in Java mapped to `baseCall` in C#, and (2) the syntactic symbol sequences encoded for each statement type in both languages, such as `StatementExpression;` in Java encoded as "EXPR SC".

Given a Java Android code block, Native-2-Native [15] extracts program identifiers to search for Swift iOS code blocks relevant to those identifiers using popular web programming resources (e.g.,

MOBILESoft'18, Gothenburg, Sweden

DOI: 10.1145/nnnnnnn.nnnnnnn

Google Code, StackOverflow, etc.). Unlike tInferer, Native-2-Native neither infers nor applies any translation rules between the languages.

In the context of the same language, various approaches infer program transformation rules by comparing the old and new versions of one or more code change examples [19–23, 30, 33]. However, these approaches rely on the source and target codebases sharing the same grammar, thus being unable to infer cross-language translation rules.

Some existing HTML5 frameworks (e.g., Sencha [13], PhoneGap [11], Appcelerator [3], React Native [12]) can automatically translate Javascript/HTML5 to Java or Swift. However, these tools require that developers follow specific JavaScript APIs rather than supporting general language-to-language translation.

1.2 Novelty and Contributions

This paper describes the design and implementation of our novel automated approach tInferer that complements prior work. We notice that the source and target codebases of native cross-platform mobile apps encode translation rules over a variety of syntax levels. Our approach captures these rules and automatically applies them to guide the porting process. Specifically, tInferer operates in two phases. In the first phase, tInferer infers rules by comparing the existing implementations in both languages; it iteratively aligns and matches Java and Swift code based on *braces* (i.e., { and }) and string similarity. As both languages are object-oriented, they share the basic syntactic components, including class declarations, method declarations, loop structures, and conditional statements. The braces that delimit these components are used identically in both languages, and thus can serve as anchors to align the code regions that are potentially semantically equivalent.

tInferer is the first approach that iteratively infers and applies both syntax and API migration rules from Java to Swift based on the minimalist developer-provided domain knowledge of language correspondence. tInferer leverages two intuitions. First, with the meaning of key arithmetic and logic operators being internalized as part of elementary math education, language designers tend to avoid redefining this meaning. Second, other operators and markers often also have historically established semantics in modern OO languages. For example, the dot operator (i.e., .) accesses object members, parentheses (i.e., `()`) delimit expressions, and braces (i.e., {}) mark code blocks. As these operators and markers have the same semantics across different languages, they can serve as anchors for tInferer, which aligns and compares the equivalent coding idioms.

Additionally, tInferer uses the Java and Swift syntax trees of matched code to determine in which order multiple operators of an expression should serve as alignment anchors. Leveraging the operator precedence commonality between languages, tInferer can iteratively find the highest matching operator in syntax trees to split code in different ways, and may infer multiple code migration rules at different levels from a single code pair. By relying on tree-based string splitting and matching, tInferer can infer more template and argument mappings than pure, delimiter-based, non-iterative approaches. tInferer works without requiring developers to manually specify the correspondence between syntactic components of different languages, or to encode syntactic structures as sequences of syntactic symbols, thus increasing the level of automation it provides.

In summary, this paper makes the following contributions:

- We designed and implemented tInferer, the *first delimiter-based iterative rule inference and application approach for automated Java-to-Swift migration*, in which delimiters include keywords, operators, and markers. This novel approach can match not only divergent grammars between languages, but also APIs defined in different libraries.
- We designed and implemented a novel way to represent transformation rules as string templates, and to maintain the rules using a database. This novel application of the database enables users to easily understand and modify the inferred rules, and even augment the database with missing rules to enhance tInferer's code migration capability.
- We conducted a comprehensive evaluation of tInferer with 3,859 real code migration examples. Our evaluation shows that tInferer was able to accurately infer and apply many translation rules for statements, expressions, and API usage. It will save the manual effort required to enforce these rules for syntax and API mappings. Hence, by automating the simple migration tasks that require mild changes, tInferer can effectively improve software quality and increase programmer productivity.

2 MOTIVATION AND APPROACH OVERVIEW

2.1 Motivating Scenario

To give an overview of our approach, we present a running example based on the charts application. Suppose that a new developer, Alex, joins a development team that maintains both Java and Swift versions of a mobile app. Figure 2(a) shows the abbreviated versions of the app's current implementation, with the Java and Swift versions depicted on the left and right, respectively. Although the two versions have similar layouts of their respective class and method declarations, they differ in the following three aspects.

- **The keyword sets.** For example, `extends` is unique to Java, while `let` only exists in Swift.
- **The statement syntaxes.** For instance, the header of `for`-loop has the format `for([ForInit]; [Expression]; [ForUpdate])` in Java; nevertheless in Swift, it becomes `for Pattern in Expression`.
- **The API usage of fields, methods, and types.** On line 6 of Figure 2(a), `mData.get(i)` in Java corresponds to `data[i]` in Swift.

Suppose that Alex is expert in Java, but is fairly new to Swift. To add a new feature to the app, Alex first would have to implement and test a new method `calcNewAngle()` in Java (see Figure 2(b)), and then manually translate the tested implementation statement-by-statement to Swift. Such manual code migration is laborious and intellectually tiresome for two reasons. First, Alex has to consistently replace language keywords and adjust program structures, while ensuring that control and data dependencies between local variables and fields remain intact. Second, Alex needs to learn how the APIs correspond between the languages to correctly translate

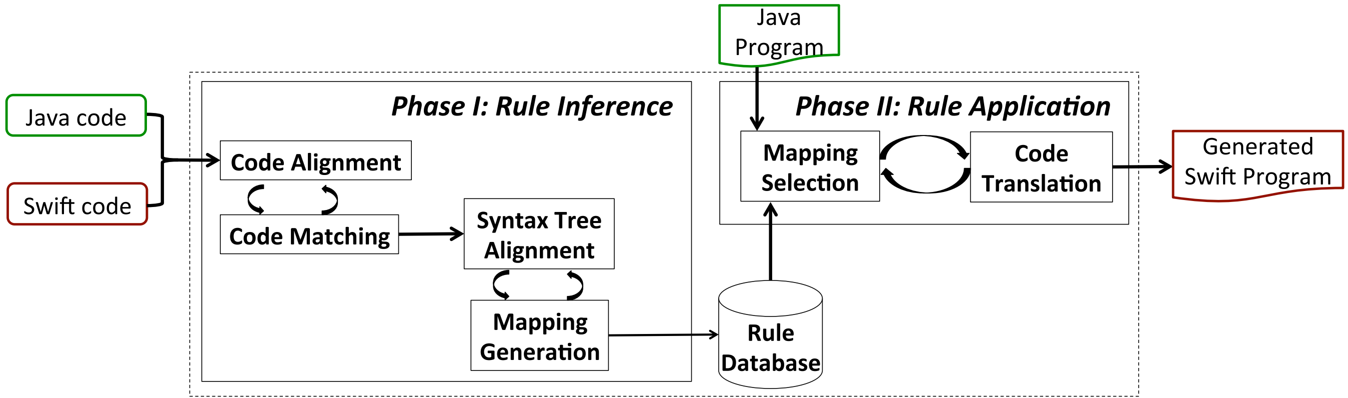

Figure 1: tInferer consists of two phases: rule inference and application. The first phase takes in both Java and Swift codebases to infer translation rules. The second phase takes in a Java program to produce a Swift implementation.

API method calls, field accesses (e.g., `data.entryCount`), or member accesses (e.g., `set[i]`) from Java to Swift.

2.2 Approach Overview

As per Figure 1, tInferer operates in two main phases: rule inference and rule application.

In **Phase I**, given a corpus of software with both Java and Swift versions provided (P_j and P_s), tInferer extracts semantically equivalent code regions by iteratively aligning and matching code. Specifically, tInferer first aligns and matches source files between P_j and P_s based on file names. If two files are named similarly, they are likely to implement the same functionalities. Among the aligned files and based on the *usage of nested braces* (i.e. { and }), tInferer then iteratively (1) aligns class declarations, method declarations, as well as statements, and (2) matches code based on the string similarity. Next, for each matched pair of statements (e.g., ¡S_j, S_s¿), tInferer parses syntax trees; it relies on keywords (e.g., `while`), operators (e.g., `=`), and markers (e.g., `,`) to align syntax subtrees. Leveraging a novel tree-traversal algorithm that matches semantically equivalent syntax tree nodes, tInferer flexibly infers the translation rules for statements, expressions, as well as API invocations, even though Java and Swift have different grammars and many divergent syntax node types. Finally, tInferer saves the rules as string templates in its database.

In **Phase II**, given a Java program, tInferer performs statement-to-statement translation to generate a translated Swift version by searching for any applicable translation rule for each statement. If one rule is applicable, tInferer transforms code accordingly. If there are multiple rules applicable, tInferer ranks the rules based on their occurrence frequencies in Phase I, and picks the most popular rule to apply. When no rule is applicable, users can either augment tInferer's database with the missing rules to automate the transformation, or manually translate code based on the tool-generated version.

To evaluate tInferer, we created a dataset based on the Java and Swift versions of four applications: charts [6, 7], antlr4-runtime [2], cardboard [4, 5], and geometry-api [8, 9]. Exploiting the iterative aligning and matching method of Phase I, we extracted 3,859 statement-level matches between the two versions of all four applications. The extracted code was used for tInferer's accuracy evaluation. To measure tInferer's in-project translation accuracy, for each project, we used 75% data for rule inference, and the other 25% for rule application. We found tInferer to correctly translate code with 76% accuracy on average. To evaluate the cross-project translation accuracy, we leveraged the data of three apps for rule inference, and the data of the fourth one for rule application. tInferer achieved 65% accuracy on average. Finally, we used one half of the extracted code for tInferer to infer rules, and the other half for both tInferer and j2swift [10] to translate code. By comparing the generated code of both tools against the original Swift version, we found that tInferer achieved 76% accuracy, which was much higher than j2swift's accuracy of 57%.

Based on the outmost brace pair of Figure 2(a), tInferer aligns lines 1-7, compares the code regions, and finds their string content to be similar. Leveraging this finding, tInferer further aligns lines 2-7 based on the intermediate brace pair, and lines 5-7 based on the innermost brace pair. It then compares code line-by-line within each aligned region. This iterative process continues until every pair of similar lines are identified. After identifying similar lines, tInferer represents each line as one or more **string templates** [29], with every template containing **abstract parameters** (`$p`) that can be concretized with **arguments** (i.e., expressions or idenfiers). tInferer maps the string templates and arguments between languages by aligning the program syntax trees based on *commonly used keywords, operators, and markers across languages*, as *the meaning of these notations seldom vary between different object-oriented (OO) languages*. For our example in Table 3, tInferer extracts string templates from both versions, and records the corresponding Java syntax node type of each template. It also records the argument mappings, such as `PieChart` and `PieChartView` shown in Table 4.

The first phase of tInferer produces a collection of migration rules, in which each rule comprises a string template and argument mappings. The second phase applies these mappings to translate Java to Swift line by line. Specifically, it coverts each Java line to a syntax tree. According to the root node's type, tInferer searches its rule database to find all matching Java templates. The found

(a) The Java and Swift versions of PieChart used for *rule inference*

```java
  //PieChart.java
1 public class PieChart extends PieChartBase<PieData> {
2   private void calcAngles() { ...
3     int entryCount = mData.getEntryCount();
4     int cnt = 0;
5     for(int i = 0; i < mData.getDataSetCount(); i++){
6       IPieDataSet set = mData.get(i);
7     ...}}}
```

```swift
//PieChartView.swift
public class PieChartView: PieChartViewBase {
  private func calcAngles() { ...
    let entryCount = data.entryCount
    var cnt = 0
    for i in 0 ..< data.dataSetCount {
      let set = data[i]
    ...}}}
```

(b) Generating Swift code from a given Java program with *rule application*

```java
  //new example
1 private void calcNewAngle(){...
2   int c = set.getDataSetCount();
3   for (int j = 1; j < c; j++) {
4     IPieDataSet elem = set.get(i);
5     mAbsoluteAngles[cnt] += elem.get();
6   ...}}
```

```swift
//translated code from templates
private func calcNewAngle() {...//translated
  let c = set.dataSetCount      //translated
  for j in 1 ..< c {            //translated
    let elem = set[i]           //translated
    mAbsoluteAngles[cnt] += elem.get()//?
  ...}}
```

Figure 2: A code migration example

No.	Java syntax node type	Java template	Swift template
1	typeDeclaration	`public class $p10 extends $p11 {...}`	`public class $p10: $p11 {...}`
2	classBodyDeclaration	`private void $p20() {...}`	`private func $p20() {...}`
3	localVarDeclStatement expression	`$p30 $p31 = $p32;` `$p33.getEntryCount()`	`let $p31 = $p32` `$p33.entryCount`
4	localVarDeclStatement	`$p40 $p41 = $p42;`	`var $p41 = $p42`
5	statement expression	`for($p50 $p51 = $p52; $p51 < $p53; $p51++) {...}` `$p54.getDataSetCount()`	`for $p51 in $p52 ..< $p53 {...}` `$p54.dataSetCount`
6	localVarDeclStatement expression	`$p60 $p61 = $p62;` `$p63.get($p64)`	`let $p61 = $p62` `$p63[$p64]`

Figure 3: String template mappings

Parameter	Argument in Java	Argument in Swift
`$p10`	`PieChart`	`PieChartView`
`$p11`	`PieChartBase<PieData>`	`PieChartViewBase`
`$p20`	`calcAngles`	`calcAngles`
`$p31`	`entryCount`	`entryCount`
`$p32`	`mData.getEntryCount()`	`data.entryCount`
`$p33`	`mData`	`data`
...	...	...

Figure 4: Argument mappings

templates are then ranked based on their occurrence frequencies in the codebase, with the most frequently used template selected for the translation. Starting with the selected Java template, TINFERER creates a Swift code fragment based on the related Swift template and relevant argument mappings.

By applying TINFERER, Alex obtains a partially translated Swift version of the application, with the lines that TINFERER could not translate simply copied over and explicitly marked (e.g., `//?` on line 5 in Figure 2 (b)). By focusing Alex's manual effort on these hard-to-translate lines, TINFERER can simplify the tedious and error-prone task of translating applications between different languages.

3 DESIGN AND IMPLEMENTATION

As shown in Figure 1, TINFERER contains two phases: rule inference and rule application. The first phase takes as input a corpus of software implemented in both Java and Swift, and iteratively aligns and matches code and syntax trees to generate string template and argument mappings. The second phase takes as input a new Java program, selects applicable template and argument mapping rules, and iteratively applies the rules to produce Swift code. In this section, we first discuss how code is aligned and matched (Section 3.1), and then describe how rules are generated (Section 3.2) and applied (Section 3.3).

3.1 Alignment and Matching of Code

Intuitively, when translating code from one OO language to another, developers are likely to convert code statement by statement. Therefore, to mimic developers' intuitive code translation practice, we designed TINFERER to infer Java-to-Swift translation rules from a corpus of software in Java and Swift by first identifying the code regions that may have statement-to-statement correspondences between the versions. To locate these regions, our approach leverages braces, line separators, and string similarity.

String Similarity-Based Source File Comparison. When a program is implemented in Java (P_j) and in Swift (P_s), TINFERER first compares the string similarity of file names in both programs using SimMetrics [14]. If the string similarity between two names is above 0.7, TINFERER considers the files (i.e., F_j and F_s) as a matching pair that may implement identical functionalities.

Code Normalization. Before further aligning and matching code for each file pair, TINFERER normalizes source code to remove comments and any unnecessary line break within a statement, and to add braces to delimit the body of compound statements (e.g., `while`-loop's body) if there is none. By standardizing code representation, tInferer ensures that the same type of statements always have the same layout and format, and thus can be processed in the same way. With more details, tInferer creates a syntax tree for each file using ANTLR [1], and implements a pretty printer to traverse the tree and to print source code in a standard way. Figure 2 (a) demonstrates the source code after normalization.

Code Region Alignment Based on Braces and Line Separators. Starting from the normalized representation, tInferer aligns code based on braces, because OO languages commonly use braces to delimit the body of class declarations, method declarations, and compound statements (e.g., `switch`-statement and `for`-loop). We use the term **braced region** to refer to the block delimited by { and } plus the code right before but on the same line as the open brace (such as class header and `if`-condition). Here the code line right before the open brace is called **header**. For our example in Figure 2, lines 1-7 in both versions are aligned in this way. By delimiting code blocks and statements with line separators and braces, we aimed to simplify the problem of inferring statement-level program syntactic mappings across languages to the problem of reasoning about mappings between similar code lines.

Code Region Matching Based On String Similarity. tInferer compares the aligned code region for string similarity. If two code regions (e.g., R_j and R_s) have at least 0.5 similarity, tInferer further aligns and matches any braced region inside them to establish finer-granularity matching. If R_j and R_s do not contain any matching inner braces, tInferer compares their code line by line. Code lines are considered to match if they have at least 0.5 similarity. As mentioned in Section 2, Java and Swift code can be different in several aspects. By using a relatively low similarity threshold (i.e., 0.5), we are able to match semantically equivalent implementations while tolerating some syntactic differences. If one line L_j in R_j matches multiple lines in R_s or vice versa, tInferer picks the line with the highest similarity score in R_s as the best match for L_j. This iterative alignment and matching process continues until every pair of similar lines is identified. Due to our code normalization, each matched line pair can be simply considered as a pair of matching statements. tInferer aims to infer translation rules from matched statements, and then to automate statement-to-statement translation by applying the inferred rules.

3.2 Syntax Tree Alignment and Mapping

For each pair of similar lines or matching statements, tInferer parses syntax trees, and aligns subtrees relying on basic language features like the commonly used keywords, operators, and markers. Our insight is that *different OO languages have similar basic language features. By using the common features as anchors, we can align distinct code fragments across languages and thus infer the underlying translation rules.* To align syntax trees and generate mappings, tInferer takes five steps. Algorithm 1 formally describes the five-step process.

To facilitate the explanation of our algorithm, we also present the ANTLR-genereated syntax trees of a matching code pair in Figure 5. Both the Java and Swift statements implement the same functionality (i.e., obtaining the size of a collection). As shown in Figure 5, ANTLR generates a separate branch for each identifier, keyword, operator, and marker. It may create a sequence of

Algorithm 1: Generating template and argument mappings

```
Input  :(L_j, L_s) /* pair of matching lines between
        Java and Swift                                 */
Output:(M_t, M_a) /* mappings of string templates and
        arguments                                      */
M_t:= ∅, M_a:= ∅;
queue_j:= ∅, queue_s:= ∅;
/* 1. initial subtree matching                         */
T_j:= getST(L_j);
T_s:= getST(L_s);
queue_j.enque(T_j);
queue_s.enque(T_s);
while !queue_j.isEmpty() do
    /* 2. operator-based substring extraction          */
    stree_j:=queue_j.deque(), stree_s:=queue_s.deque();
    op_j:= getHighestOp(stree_j)
    op_s:= getHighestOp(stree_s);
    tmpL_j:=getString(stree_j);
    tmpS_j:=getString(stree_s);
    if op_j ! = op_s then
        if op_j == ";" then
            specialMatch(tmpL_j, tmpS_j, M_t, M_a);
        end
        else
            flatMatch(tmpL_j, tmpS_j, M_t, M_a);
        end
    end
    else
        strs_j:= split(tmpL_j, op_j), strs_s:= split(tmpS_j, op_s);
        /* 3. substring comparison                     */
        foreach String s_j ∈ strs_j do
            s_s = findBestMatch(strs_s);
            if s_s! = null then
                M_a:=M_a ∪ (s_j, s_s);
                strs_s = strs_s- s_s;
            end
        end
    end
    /* 4. template generation                          */
    parameterize(L_j, L_s, M_a);
    template_j:=moreParameterize(tmpL_j);
    template_s:=moreParameterize(tmpS_j);
    javaNodeType := stree_j.getNodeType();
    M_t := M_t ∪ (javaNodeType, template_j, template_s);
    /* 5. mapping inference for substrings             */
    foreach (s_j, s_s) ∈ M_a do
        if !isLeaf(s_j, stree_j) && !isLeaf(s_s, stree_s) then
            t_j:=getST(s_j);
            t_s:=getST(s_s);
            queue_j.enque(t_j);
            queue_s.enque(t_s);
        end
    end
end
```

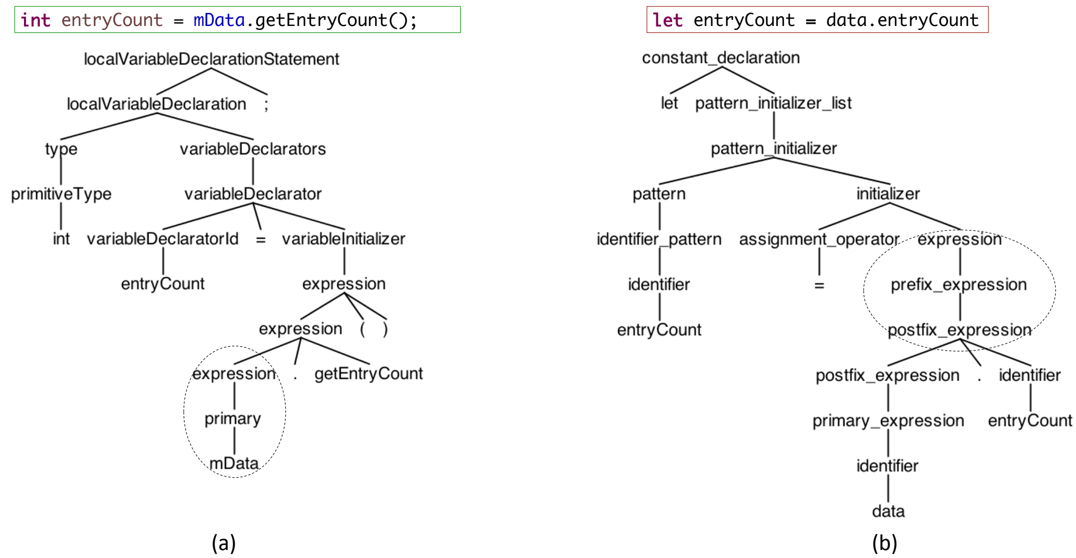

Figure 5: Syntax trees of a pair of matching lines

single-child inner nodes before producing a leaf node or several branches (e.g., `mData` and `data.entryCount` as circled with dashed lines). Although the two statements look similar, their syntax trees differ in terms of the tree heights, node types, and tree structures. Semicolons are mandatory in Java but optional in Swift.

Step 1: Initial Subtree Matching. Given the matching code pair, tInferer traverses their syntax trees to find the lowest level of subtree pairs that reflect the string matching while ignoring semicolons. As shown in Figure 5 (a), the inner node `localVariableDeclaration` in Java corresponds to `int entryCount =mData.getEntryCount()`, while the node `constant _declaration` in Swift corresponds to `let entryCount =data.entryCount`. Based on the similar strings, we match these two subtrees (`localVariableDeclaration`, `constant_-declaration`), and adds the subtrees to two separate queues—$queue_j$ and $queue_s$—for further processing.

Step 2: Operator-Based Substring Extraction. Given two matching subtrees: $stree_j$ and $stree_s$, tInferer searches for the highest operator in each syntax tree (i.e., op_j and op_s). If the operators are the same, tInferer further extracts expressions and identifiers by splitting each string based on the matching operator, whitespace, and semicolon, and by excluding structure-relevant keywords from the substring set. In our research, we classify keywords into two types: structure-relevant vs. structure-irrelevant, and treat them differently. The structure-relevant keywords (e.g., `for`) are inherent in program structures. They do not vary with the content put into the structures, and thus *should be included as parts of the generated templates.* The structure-irrelevant keywords (e.g., `int`) are not closely bound to any syntactic component, and thus does not indicate the program structure. They are replaceable by other identifiers, and *should be parameterized away instead of included* when generating templates.

For our exemplar statement `int entryCount=mData.getEntryCount()`, the highest operator is =, which matches the highest operator in the Swift code. Based on this operator and whitespace, we can split the Java code into three substrings: `int`, `entryCount`, and `mData.getEntryCount()`. None of these substrings is a structure-relevant keyword, so they are all included into the resulting substring set $strs_j$. Similarly, tInferer also splits the Swift statement `let entryCount=data.entryCount` into three substrings: `let`, `entryCount`, and `data.entryCount`. However, as the keyword `let` is structure-relevant, tInferer excludes it from the resulting substring set $strs_s$.

Step 3: Substring Comparison. Between the two extracted substring sets—$strs_i$ and $strs_j$, tInferer exhaustively compares strings pair-by-pair to find the best match for each string, and to reveal the correspondence between different parts of the code pair. The default similarity threshold is 0.5, meaning that if the similarity between two strings is less than 0.5, they are considered dissimilar. In our example, `entryCount` exists in both versions and matches, while `mData.getEntryCount()` and `data.entryCount` are similar and matched. However, `int` does not match anything, indicating that this identifier is used only in the Java version, without being translated to Swift. Therefore, all matched substrings are saved as argument mappings in a database, as shown in Table 4. ***Step 4: Template Generation.*** For the given code pair, tInferer creates templates by consistently replacing each pair of matched substrings with the same parameter, and by replacing each unmatched substring with a unique parameter. For our example, tInferer infers the Java template `$p30 $p31 = $p32;` and the Swift template `let $p31 = $p32`.This template pair and the Java syntax node type are saved as a template mapping in the database, as shown in Table 3.

Step 5: Mapping Inference for Substrings. The template and argument mappings generated so far only describe statement-level mappings, without showing how expressions can be structured differently between languages. Such coarse-grained mappings only allow tInferer to restructure statements by moving Java expressions around, but do not support further translations of Java expressions to Swift ones. To enable expression-level translations, tInferer continues comparing matched substrings, and generates fine-grained mappings by repeating Steps 1-4 iteratively. For our example, the further iteration of Steps 1-4 on substrings `mData.getEntryCount()` and `data.entryCount` produces an extra template mapping—(`expression`, `$p33.getEntryCount()`, `$p33.entryCount`), and an additional argument mapping (`$p33, mData, data`). With the template mapping, we know how to translate a Java method API invocation (i.e., $p33.getEntryCount()) to a Swift field API access (i.e., $p33.entryCount).

Although our algorithm is intuitively explained with a pair of variable declaration statements, the algorithm also contains special logic to effectively handle compound statements (e.g., `for`-loop) and code pairs without matching operators (e.g., `a+b` vs. `sum(a, b)`). In particular, as shown in Figure 6, the syntax tree of a compound statement (e.g., `for`-loop) may correspond to the statement itself together with those statements under the structure (e.g., statements inside the `for`-loop body). In such scenarios, Step 1 initializes subtree matching for the whole compound statement, while Steps 2-5 only focus on the header's subtrees ignoring the statements contained by the body. When semicolons are used in the header of Java `for`-loop, tInferer implements a separate match method *specialMatch*(...) to specially treat semicolons as delimiters used in the resulting inferred template. Additionally, in Step 2, if two syntax trees have no matching highest-level operator (e.g., `a+b` vs. `sum(a, b)`), tInferer implements $flatMatch(...)$ to simply split each string based on all operators, markers, and whitespace.

3.3 Template Selection and Code Translation

If we consider the above rule inference process as iteratively replacing concrete substrings with abstract parameters to generate mappings, then the rule application phase can be considered as the reverse process. It iteratively selects mappings to generate code by replacing abstract parameters with concrete substrings. Therefore, some functions mentioned in Section 3.2 can be reused in this phase.

Given a Java code line to translate (e.g., `for(int j=1;j<c;j++){}`, tInferer first locates the lowest subtree that matches the code (*getST*(...)), and then selects related template mappings based on the subtree's node type. Figure 6 presents the syntax tree found for the above exemplar Java code. Based on the syntax node type (`statement`), tInferer queries its database to get all relevant template mappings. Among all the mappings shown in Table ??, there is only one relevant mapping as shown below, which is selected to translate this code.

```
(statement,for($p50 $p51 = $p52; $p51 < $p53; $p51++) {...},
                                    for $p51 in $p52 ..< $p53 {...})
```

Code translation involves two parts: string-template matching and argument replacement. With template mappings selected based on a syntax node type, tInferer tentatively matches the given Java code with each Java template to check which template mapping is applicable. If multiple mappings are applicable, tInferer picks the one that occurs most in the rule inference phase. For our example, the Java code matches the Java template in the above mapping (3.3). Therefore, tInferer identifies the following correspondence between concrete substrings and abstract parameters accordingly:

```
(int, $p50), (j, $p51), (1, $p52), (c, $p53)
```

According to the template mapping (3.3), tInferer detects that `$p51`, `$p52`, and `$p53` are reused in the Swift template. It then queries the database for argument mappings related to any of these concrete substrings: `j`, `1`, and `c`. If there is such an argument mapping, tInferer simply uses the corresponding Swift substring to generate code; otherwise, tInferer reuses the Java substring for code generation. In our example, there is no argument mapping found, so tInferer translates code by replacing parameters used in the Swift template with corresponding Java substrings, producing the following Swift-style string: `for(j in 1 ..< c){`.

Finally, tInferer checks whether each extracted Java substring (i.e., `int`, `j`, `1`, and `c`) corresponds to a leaf node or inner node in the original syntax tree. If a substring corresponds to a leaf node, the substring is an identifier, and does not need any further conversion. However, if a substring corresponds to an inner node, tInferer leverages the inner node's type to query the database, and to iteratively convert Java expressions to Swift ones. The process continues until every Java expression is converted, or until there is no applicable template mapping for translation. In our example, since all Java substrings are identifiers, tInferer does not need to convert any expressions after producing the Swift-style string mentioned above.

A naïve non-iterative alternative rule inference and application algorithm. To generate template and argument mappings from the matching statements between Java and Swift, a naïve non-iterative alternative would be to simply extract substrings based on all operators, markers, whitespace, and keywords, and then to establish mappings between the two substring sets. Although this approach can generate some mappings, the applicability of the inferred mappings is limited. For instance, given two similar strings: `c=a+b;` vs. `c=sum(a, b)`, only one template mapping can be inferred in this way: (`localVarDeclStat $p0=$p1+$p2; $p0=sum($p1, $p2)`). This mapping does not enable tInferer to convert `d=a+b+c` to `d=sum(sum(a, b), c)`. However, with our syntax tree-based iterative template inference algorithm, tInferer can correctly translate the expression by iteratively applying two inferred template mappings: (`localVarDeclStat, $p0=$p1;, $p0=$p1`), (`expression, $p1+$p2, sum($p1, $p2)`).

4 EVALUATION

This section presents our evaluation data set (Section 4.1), the accuracy metric (Section 4.2), and the three experiments we conducted on the dataset (Section 4.3, 4.4, and 4.5).

4.1 Dataset

To evaluate how effectively tInferer can infer and apply rules to translate Java code to Swift, we collected four subject applications: charts [6, 7], antlr4-runtime [2], cardboard [4, 5], and geometry-api [8, 9], each of which has both Java and Swift implementations. A brief description of our subjects is as follows. charts is a chart/graph view library that supports different kinds of charts, including line charts, bar charts, and pie charts. antlr4-runtime is a component

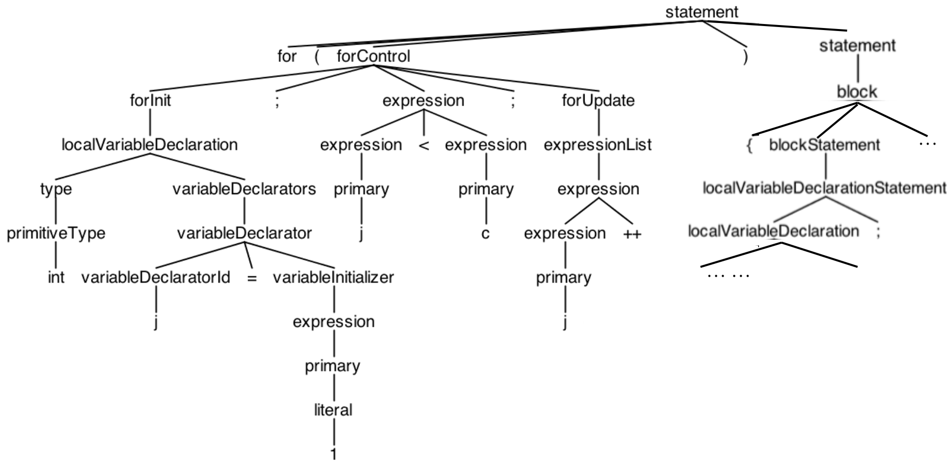

Figure 6: In this Java syntax tree of `for(int j=1; j<c; j++)...`, we use ellipses ("…") to succinctly represent content in the loop body.

of ANTLR, which is a parser generator for reading, processing, executing, or translating structured text or binary files. ANTLR provides multiple versions of its runtime component in different languages. cardboard is the Google cardboard virtual reality (VR) toolkit library, which simplifies common VR development tasks, such as lens distortion correction and head tracking. geometry-api provides APIs for simple geometries, spatial operations, and topological relationship tests.

By aligning the two versions of each application in the way mentioned in Section 3.1, we identified a subset of the code whose implementation can be intuitively matched across languages. Table 1 shows the lines of code (LOC) for both versions of each subject, and the LOC with successful alignment. In total, there are 3,859 LOC aligned between the two versions of programs. We used these 3,859 aligned code pairs as the data set in our evaluation. These code pairs cover syntax components that include class, field, and method declarations, compound statements, and simple statements.

Table 1: Dataset of cross-platform applications

Project	Java LOC	Swift LOC	Aligned LOC
charts	29,861	22,428	**2,507**
antlr4-runtime	24,617	24,603	**1,023**
cardboard	7,006	3,279	**176**
geometry-api	105,887	993	**153**

According to Table 1, geometry-api has the largest Java codebase but the smallest Swift codebase, with only 153 LOC aligned between the two versions. This discrepancy is due to the Swift version only partially implementing the functionalities of the Java version. charts has a relatively large code size in both Java and Swift versions, and contains the largest LOC value (i.e., 2,507) for the aligned code.

Table 2 demonstrates the top 10 most frequent template mappings inferred from the aligned code of charts. The top-1 most frequent template mapping corresponds to local variable declarations. The only difference between the two templates is that Swift code does not need a type identifier when declaring a variable; instead, it requires using the `var` keyword. The 2^{nd} and 7^{th} most frequent template mappings have identical Java parts but slightly different Swift parts. This peculiarity indicates that when translating some types of statements (e.g., `if`-statements), developers actually followed more than one translation strategy, as guided by their individual coding styles or preferences. However, manually defined rules describe at most one strategy for each kind of translated statements. By precisely capturing the alternative implicit translation rules that were actually applied in manual code translation, tInferer can be flexibly configured to favor any of the alternative template mappings for a translation task.

Template mappings ranked 3^{rd}, 4^{th}, 7^{th}, and 8^{th} have identical Java and Swift counterparts. One may deem mappings like these unnecessary, but in fact they serve two purposes. First, when inferring translation rules, these mappings work as anchors that align heterogeneous syntax trees between Java and Swift, and further reveal lower-level structural and content mappings. For example, given `a=b=c` in Java and `a=(b=c)` in Swift, by leveraging the 4^{th} mapping when comparing two syntax trees, tInferer can correctly align the first assignment operator in both versions, and infer the following argument mappings: (`$p0, a, a`), (`$p1, b=c, (b=c)`). Second, when translating code, these mappings help correctly split Java code, enabling separate translation of each substring. For example, based on the 7^{th} and 8^{th} mappings, tInferer can correctly translate `if (!m.aname())` to `if (!m.sname())`, when there is an argument mapping ($p0, `m.aname()`, `m.sname()`) in its database.

4.2 Accuracy

To measure tInferer's effectiveness, we define **accuracy** as *the percentage of lines that tInferer has translated correctly.* To decide whether a line of code is translated correctly, we compare the tool-generated code with the original Swift version of applications. The translation is considered *correct* if one of the following criteria applies:

- the translated version is identical to the human-translated version (oracle);
- the translated version has no syntax error and is syntactically similar to the oracle, with the differences confined to the usage of identifiers; and

Table 2: Top 10 most frequent template mappings inferred from charts

Rank	Java syntax type	Java template	Swift template	Cnt
1	localVarDeclStmt	$p2 $p0 = $p1;	var $p0 = $p1	90
2	statement	if ($p0) {...}	if $p0 {...}	70
3	importDecl	import $p0;	import $p0	69
4	statement	$p0 = $p1;	$p0 = $p1	68
5	classBodyDecl	public $p0 $p1 ( ) {...}	public var $p1 : $p0 {...}	65
6	statement	return $p0;	return $p0	63
7	statement	if ($p0) {...}	if ($p0) {...}	37
8	expression	! $p0	! $p0	29
9	classBodyDeclStmt	private $p2 $p0 = $p1;	private var $p0 = $p1	24
10	classBodyDecl	public $p0 $p1($p2 $p3) {...}	public func $p1($p3: $p2) -> $p0 {...}	19

- the translated version is syntactically valid and semantically equivalent to the oracle, while adhering to a different syntax.

We conducted three experiments to measure tInferer's accuracy. In the first experiment, each subject's aligned code is partitioned into the inference and application subsets. We used the inference subset to infer translation rules, and the application subset to apply them, thereby evaluating the accuracy rate of tInferer's in-project translation (Section 4.3). In the second experiment, we followed a similar process by inferring rules from three subjects, and applying them to the remaining one, thereby assessing the accuracy of tInferer's cross-project translation (Section 4.4). Finally, in the third experiment, we compared the output produced by tInferer to that produced by j2swift, a state-of-the-art Java-to-Swift translation tool (Section 4.5), thereby assessing the respective accuracy rates for both tools.

4.3 In-Project Translation

For each subject, we first identified all source files containing any aligned code. Then we used the aligned code in 75% of these files as the inference set, and the aligned code in the other 25% files as the application set. Table 3 shows tInferer's accuracy for the application set in each subject. As both Java and Swift have four common syntactic components: type declaration (TypeDecl), method declaration (MethodDecl), field declaration (FieldDecl), and statement, Table 3 presents the accuracy rate for each component.

Table 3: In-project code translation results

Common Type	charts	antlr4	cardboard	geometry-api
TypeDecl	**73%**(27/37)	**88%**(7/8)	**100%**(3/3)	**0%**(0/1)
MethodDecl	**80%**(34/42)	**78%**(34/45)	**70%**(7/10)	**36%**(4/11)
FieldDecl	**75%**(24/32)	**88%**(42/48)	-	-
Statement	**83%**(196/235)	**92%**(146/159)	**42%**(5/12)	**36%**(4/11)
Average	**81%**(246/346)	**88%**(229/260)	**60%**(15/25)	**35%**(8/23)

The overall accuracy is 76%(498/654) on average. cardboard and geometry-api lack data for FieldDecl type mainly because developers did not intuitively translate the field declarations across languages. Among the four common types, Statement has the highest average accuracy (84%), while MethodDecl obtains the lowest one (73%). This discrepancy is caused by the fact that there are more template variants for method declaration headers than statements. The number and sequential order of parameters and modifiers (e.g., `final` and `static`) can produce numerous formats of method declaration headers, making it harder to match a given concrete header with already inferred templates.

charts and antlr4-runtime show higher overall accuracy (81% and 88%) than cardboard and geometry-api (60% and 35%). We noticed that the owners of the first two applications (charts and antlr4-runtime) maintain both the Java and Swift versions, while the other two applications' different versions (cardboard and geometry-api) are owned by different people. This observation may indicate that when the same developer maintains both versions of an application, she is likely to manually port one language implementation to the other (i.e., Java to Swift). With such manually translated applications, tInferer can effectively infer and apply the intuitive code translation rules with high accuracy. However, when the Java and Swift versions are maintained by different developers, these versions tend to be built independently instead of being manually ported from one to another. Therefore, it is not surprising that tInferer achieves lower accuracy when we compare its translation with a separately built Swift version.

4.4 Cross-Project Translation

In addition to inferring and applying translation rules within the same project, we also inferred rules from three subjects, and then applied the inferred rules to a fourth subject. Compared with the in-project translation, such cross-project translation may be unable to identify sufficient project-specific mappings for the target Java project. However, this translation strategy can better fit the real usage scenarios, in which developers create a brand new Swift project by using tInferer to automatically translate their Java code.

Table 4 shows the cross-project translation results. The overall accuracy is 65%(644/994) on average, which is lower than the in-project translation accuracy reported in Section 4.3. Specifically, we obtained a much lower accuracy result for charts (51% vs. 81%). This significant accuracy decrease occurs because charts' training data of cross-project translation is much smaller than that of in-project translation, and insufficient training data weakens the translation capability. antlr-runtime's accuracy rate of 69% is much lower

Table 4: Cross-project code translation results

Common type	charts	antlr4-runtime	cardboard	geometry-api
TypeDecl	**47%**(10/21)	**86%**(36/42)	**100%**(8/8)	**100%**(2/2)
MethodDecl	**60%**(39/65)	**66%**(80/122)	**56%**(19/34)	**62%**(38/61)
FieldDecl	**42%**(34/81)	**57%**(98/174)	-	-
Statement	**56%**(73/122)	**84%**(128/152)	**93%**(39/42)	**80%**(40/50)
Average	**51%**(156/307)	**69%**(342/490)	**78%**(66/84)	**71%**(80/113)

than the 88% of the in-project translation, despite the substantially enlarged training set for cross-project translation. This is because there are many mappings specific to antlr-runtime (e.g., project-specific method calls), which are not inferable from other subjects' data. Both cardboard and geometry-api achieve higher accuracy for cross-project than in-project translations, because the training data from other projects manifests a more comprehensive set of translation rules.

4.5 Comparison with j2swift

j2swift [10] is a state-of-the-art Java-to-Swift syntax converter. It leverages ANTLR to create a parse tree for Java, and implements manually defined syntax conversion rules to generate Swift code while walking the parse tree. The documentation claims that j2swift finishes 80% translation tasks for simple Java code. We used one half of the files with aligned data for tInferer to infer rules, and the other half of the files to evaluate the code translation accuracy of both tInferer and j2swift. Table 5 shows that tInferer outperforms j2swift for each common type. The average accuracy of tInferer is 76%, which is much higher than tInferer's 57% accuracy rate. This observation is unsurprising, because tInferer flexibly infers both template and argument mappings, while j2swift hard-codes only some template mappings. When translating Java code, tInferer has more template mappings and argument mappings to apply than j2swift. Consider translating `if(set.getEntryCount() >max.getEntryCount())` to `if set.entryCount>max.entryCount`. Without encoding the domain knowledge of mapping member APIs (`$p.getEntryCount()` vs. `$p.entryCount`), j2swift can only copy the original code to Swift code without translating it. In comparison, tInferer can translate this code correctly due to its inferred rules.

Table 5: tInferer vs. j2swift

Common type	tInferer	j2swift
TypeDecl	**65%**(37/57)	**40%**(23/57)
MethodDecl	**58%**(71/122)	**55%**(67/122)
FieldDecl	**70%**(150/214)	**38%**(82/214)
Statement	**83%**(471/565)	**66%**(372/565)
Average	**76%**(729/958)	**57%**(544/958)

5 CONCLUSIONS AND FUTURE WORK

As native cross-platform mobile apps have become an industry standard, their development remains challenging. We presented tInferer that facilitates the porting of such apps between different platforms. The data-driven nature of tInferer causes its effectiveness to grow with the number of codebases available for rule inferencing. In our evaluation, even with the limited training data, tInferer clearly outperformed j2swift in terms of translation accuracy. As a future work, we plan to enhance tInferer to support translation involving code refactoring, to further improve tInferer's translation accuracy, and to extend it to handle other inter-language translation tasks. As major mobile platforms keep competing for market dominance, mobile developers will continue translating their apps across languages, and our approach can streamline this non-trivial process.

AVAILABILITY

The source code of tInferer described in the paper can be downloaded from this website: https://git.cs.vt.edu/ankijin/tinferer.

ACKNOWLEDGEMENTS

This research has been supported by the National Science Foundation through the Grants #1649583, 1717065, 1650540 and 1565827.

REFERENCES

[1] ANTLR. http://www.antlr.org.
[2] antlr4-runtime. https://github.com/antlr/antlr4/tree/master/runtime.
[3] Appcelerator. http://www.appcelerator.com/.
[4] cardboard for Java. https://github.com/rsanchezsaez/cardboard-java.
[5] cardboard for Swift. https://github.com/nzff/cardboard-swift.
[6] charts in Java. https://github.com/PhilJay/MPAndroidChart.
[7] charts in Swift. https://github.com/danielgindi/Charts.
[8] geometry-api for Java. https://github.com/esri/geometry-api-java.
[9] geometry-api for Swift. https://github.com/eito/geometry-api-swift.
[10] j2swift. https://github.com/patniemeyer/j2swift.
[11] PhoneGap. https://build.phonegap.com.
[12] React Native. https://facebook.github.io/react-native/.
[13] Sencha. https://www.sencha.com.
[14] SimMetrics. https://github.com/mpkorstanje/simmetrics.
[15] Antuan Byalik, Sanchit Chadha, and Eli Tilevich. Native-2-Native: Automated cross-platform code synthesis from web-based programming resources. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 99–108, New York, NY, USA, 2015. ACM. doi:10.1145/2814204.2814210.
[16] M. El-Ramly, R. Eltayeb, and H. A. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, 2006.
[17] Ahmed E. Hassan and Richard C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 2005.
[18] Java2CSharp. http://sourceforge.net/projects/j2cstranslator/.
[19] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.
[20] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. Does automated refactoring obviate systematic editing? In *ICSE*, 2015.
[21] Na Meng, Miryung Kim, and Kathryn McKinley. Sydit: Creating and applying a program transformation from an example. In *ESEC/FSE'11, joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 440–443, 2011.

[22] Na Meng, Miryung Kim, and Kathryn McKinley. Lase: Locating and applying systematic edits. In *ICSE*, pages 502–511, 2013.

[23] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[24] M. Mossienko. Automated Cobol to Java recycling. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, 2003.

[25] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[26] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[27] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.

[28] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392, New York, NY, USA, 2009. ACM. `doi:http://doi.acm.org/10.1145/1595696.1595767`.

[29] Terence Parr. The StringTemplate library, 2017.

[30] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE Press, 2017.

[31] Harry M. Sneed. Migrating from COBOL to Java. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010.

[32] K. Yasumatsu and N. Doi. SPiCE: a system for translating Smalltalk programs into a C environment. *IEEE Transactions on Software Engineering*, 1995.

[33] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 740–751, 2017.

[34] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204. ACM, 2010.