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 \textsc{t\textsuperscript{in}f\textsuperscript{er}}, a novel approach that iteratively infers syntactic transformation rules and API mappings from Java to Swift. Given a software corpus in both languages, \textsc{t\textsuperscript{in}f\textsuperscript{er}} first identifies the semantically equivalent code based on braces and string similarity. For each pair of similar code segments, \textsc{t\textsuperscript{in}f\textsuperscript{er}} then creates syntax trees of both languages, leveraging the \textit{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. \textsc{t\textsuperscript{in}f\textsuperscript{er}} represents inferred rules as string templates, stored in a database, to translate code from Java to Swift. We evaluated \textsc{t\textsuperscript{in}f\textsuperscript{er}} 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, \textsc{t\textsuperscript{in}f\textsuperscript{er}} outperforms in accuracy j2swift, a state-of-the-art Java-to-Swift conversion tool. As native cross-platform mobile apps grow in popularity, \textsc{t\textsuperscript{in}f\textsuperscript{er}} can shorten their time to market by automating the tedious and error-prone task of source-to-source translation.

ACM Reference format:
DOI: 10.1145/mmmnnn.mmmnnn

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 groups [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 $\text{SuperCall}$ in Java mapped to $\text{baseCall}$ in C\#, and (2) the syntactic symbol sequences encoded for each statement type in both languages, such as StatementExpression; in Java encoded as "$\text{EXPR}$."

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.,
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 pro-
gram 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 shar-
ning the same grammar, thus being unable to infer cross-language
translation rules.

Some existing HTML5 frameworks (e.g., Sencha [13], Phone-
Gap [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 migra-
tion 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 de-
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 ap-
proach 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 ap-
ply many translation rules for statements, expressions, and
API usage. It will save the manual effort required to en-
force 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 exam-
ple 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 ver-
sions 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 consis-
tently 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
API method calls, field accesses (e.g., data.entryCount), or member accesses (e.g., set(1)) from Java to Swift.

2.2 Approach Overview

As per Figure 1, TÍNFERER 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 \((P_J \text{ and } P_S)\), TÍNFERER extracts semantically equivalent code regions by iteratively aligning and matching code. Specifically, TÍNFERER 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 \((i.e., \{ \text{ and } \})\), TÍNFERER 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 \((e.g., S_j, S_s)\), TÍNFERER parses syntax trees; it relies on keywords \((e.g., \text{white})\), operators \((e.g., \text{.})\), and markers \((e.g., \text{)}\) to align syntax subtrees. Leveraging a novel tree-traversal algorithm that matches semantically equivalent syntax tree nodes, TÍNFERER 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, TÍNFERER saves the rules as string templates in its database.

In Phase II, given a Java program, TÍNFERER 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, TÍNFERER transforms code accordingly. If there are multiple rules applicable, TÍNFERER 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 TÍNFERER’s database with the missing rules to automate the transformation, or manually translate code based on the tool-generated version.

To evaluate TÍNFERER, 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 TÍNFERER’s accuracy evaluation. To measure TÍNFERER’s in-project translation accuracy, for each project, we used 75% data for rule inference, and the other 25% for rule application. We found TÍNFERER 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. TÍNFERER achieved 65% accuracy on average. Finally, we used one half of the extracted code for TÍNFERER to infer rules, and the other half for both TÍNFERER and j2swift \([10]\) to translate code. By comparing the generated code of both tools against the original Swift version, we found that TÍNFERER achieved 76% accuracy, which was much higher than j2swift’s accuracy of 57%.

Based on the outmost brace pair of Figure 2(a), TÍNFERER aligns lines 1-7, compares the code regions, and finds their string content to be similar. Leveraging this finding, TÍNFERER 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, TÍNFERER represents each line as one or more string templates \([29]\), with every template containing abstract parameters \((p)\) that can be concretized with arguments \((i.e., \text{expressions or identifiers})\). TÍNFERER 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, TÍNFERER 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 TÍNFERER 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 converts each Java line to a syntax tree. According to the root node’s type, TÍNFERER searches its rule database to find all matching Java templates. The found
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 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 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 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 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).
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 \( R_k \)) 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 \( R_k \) 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 \( R_k \) or vice versa, TInferer picks the line with the highest similarity score in \( R_k \) 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-generated 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: \((L_j, L_k)/* pair of matching lines between Java and Swift */\)

Output: \((M_t, M_a)/* mappings of string templates and arguments */\)

\(M_t := \emptyset, M_a := \emptyset;\)

queue\(_j\) := \emptyset, queue\(_k\) := \emptyset;  
/* 1. initial subtree matching */

\(T_j := \text{getST}(L_j) ;\)

\(T_k := \text{getST}(L_k) ;\)

queue\(_j\).enqueue\((T_j) ;\)

queue\(_k\).enqueue\((T_k) ;\)

while queue\(_j\).isNotEmpty() do

\(s_j := \text{queue}_j\) .dequeue() \(, stree_j := \text{ QUEUE } .\text{deque() ; \} ;\)

\(op_j := \text{getHighestOp(stree}_j);\)

\(op_k := \text{getHighestOp(stree}_k);\)

\(tmpL_j := \text{getString(stree}_j);\)

\(tmpS_j := \text{getString(stree}_k);\)

if \(op_j \neq op_k\) then

\( \text{if op}_j \neq "\" then \}

\(s_j := \text{StdConfStringMatch}(tmpL_j, tmpS_j, M_t, M_a);\)

\(\} \text{end else } \}

\(s := \text{getBestMatch}(tmpL_j, tmpS_j, M_t, M_a);\)

\(\} \text{end end}

else

\(str_j := \text{split}(tmpL_j, op_j), str_s := \text{split}(tmpS_j, op_s);\)

/* 3. substring comparison */

\(\text{foreach String } s_j \in str_j \text{ do }\)

\(s := \text{findBestMatch}(str_j);\)

\(\text{if } s \neq \text{null then }\)

\(M_a := M_a \cup (s_j, s);\)

\(str_j := str_j - s_j;\)

\(\} \text{end end end}

/* 4. template generation */

\(\text{parameterize}(L_j, L_k, M_a);\)

\(\text{template}_j := \text{moreParameterize}(tmpL_j);\)

\(\text{template}_s := \text{moreParameterize}(tmpS_j);\)

\(\text{javaNode} \times \text{node} := \text{stree}_j . \text{getNode} \times \text{type}();\)

\(M_t := M_t \cup (\text{javaNode} \times \text{node}, \text{template}_j, \text{template}_s);\)

/* 5. mapping inference for substrings */

\(\text{foreach } (s_j, s_k) \in M_a \text{ do }\)

\(\text{if } ! \text{isLeaf}(s_j, \text{stree}_j) \& \& ! \text{isLeaf}(s_k, \text{stree}_k) \text{ then }\)

\(T_j := \text{getST}(s_j);\)

\(T_k := \text{getST}(s_k);\)

queue\(_j\).enqueue\((T_j);\)

queue\(_k\).enqueue\((T_k);\)

end end
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 \( r \)—for further processing.

**Step 2: Operator-Based Substring Extraction.** Given two matching subtrees: \( stree_j \) and \( stree_r \), TINFERER searches for the highest operator in each syntax tree (i.e., \( op_j \) and \( op_r \)). 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., 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 \( \text{int entryCount} = \text{mData.getEntryCount}() \), 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 \( strs_r \).

**Step 3: Substring Comparison.** Between the two extracted substring sets—\( strs_j \) and \( strs_r \), 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 \texttt{TNIFER}\textsubscript{E}r 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, \texttt{TNIFER}\textsubscript{E}r 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 \texttt{mData.getEntryCount()} and \texttt{data.entryCount} produces an extra template mapping—(expression, \texttt{$\$p33$.getEntryCount()}, \texttt{$\$p33$.entryCount}), and an additional argument mapping (\texttt{$\$p33$, mData, data}). With the template mapping, we know how to translate a Java method API invocation (i.e., \texttt{$\$p33$.getEntryCount()}) to a Swift field API access (i.e., \texttt{$\$p33$.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., \texttt{a\texttt{+}b} vs. \texttt{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, \texttt{TNIFER}\textsubscript{E}r implements a separate match method \texttt{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., \texttt{a\texttt{+}b} vs. \texttt{sum(a, b)}), \texttt{TNIFER}\textsubscript{E}r implements \texttt{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., \texttt{for(int j=1;j<i;j++)}), \texttt{TNIFER}\textsubscript{E}r first locates the lowest subtree that matches the code \texttt{(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 \texttt{(statement)}, \texttt{TNIFER}\textsubscript{E}r 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.

\begin{verbatim}
{statement, for($\$p50$, $\$p51$ = $\$p52$; $\$p51$ < $\$p53$; $\$p51$++) {...},
  for $\$p51$ in $\$p52$ ... < $\$p53$ {...})
\end{verbatim}

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

\begin{verbatim}
(int, $\$p50$), (j, $\$p51$), (1, $\$p52$), (c, $\$p53$)
\end{verbatim}

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

Finally, \texttt{TNIFER}\textsubscript{E}r checks whether each extracted Java substring (i.e., \texttt{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, \texttt{TNIFER}\textsubscript{E}r 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, \texttt{TNIFER}\textsubscript{E}r does not need to convert any expressions after producing the Swift-style string mentioned above.

### A naive non-iterative alternative rule inference and application algorithm.

To generate template and argument mappings from the matching statements between Java and Swift, a naive 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 \texttt{c=a\texttt{+}b; vs. c=sum(a, b)}, only one template mapping can be inferred in this way: \texttt{(localVarDeclStat $\$p0$=$\$p1$+$$\$p2$; $\$p0$=sum($\$p1$, $\$p2$))}. This mapping does not enable \texttt{TNIFER}\textsubscript{E}r to convert \texttt{d=a\texttt{+}b+c} to \texttt{d=sum(a, b, c)}. However, with our syntax tree-based iterative template inference algorithm, \texttt{TNIFER}\textsubscript{E}r can correctly translate the expression by iteratively applying two inferred template mappings: \texttt{(localVarDeclStat, $\$p0$=$\$p1$; $\$p0$=$\$p1$), (expression, $\$p1$+$$\$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 \texttt{TNIFER}\textsubscript{E}r 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...
Table 1: Dataset of cross-platform applications

<table>
<thead>
<tr>
<th>Project</th>
<th>Java LOC</th>
<th>Swift LOC</th>
<th>Aligned LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>charts</td>
<td>29,861</td>
<td>22,428</td>
<td>2,507</td>
</tr>
<tr>
<td>antlr4-runtime</td>
<td>24,617</td>
<td>24,603</td>
<td>1,023</td>
</tr>
<tr>
<td>cardboard</td>
<td>7,006</td>
<td>3,279</td>
<td>176</td>
</tr>
<tr>
<td>geometry-api</td>
<td>105,887</td>
<td>993</td>
<td>153</td>
</tr>
</tbody>
</table>

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\textsuperscript{nd} and 7\textsuperscript{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, T\textsc{inferrer} can be flexibly configured to favor any of the alternative template mappings for a translation task.

Template mappings ranked 3\textsuperscript{rd}, 4\textsuperscript{th}, 7\textsuperscript{th}, and 8\textsuperscript{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\textsuperscript{th} mapping when comparing two syntax trees, T\textsc{inferrer} 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\textsuperscript{th} and 8\textsuperscript{th} mappings, T\textsc{inferrer} 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 T\textsc{inferrer}’s effectiveness, we define accuracy as the percentage of lines that T\textsc{inferrer} 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

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 intuitionally 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.

4.2 Accuracy

To measure T\textsc{inferrer}’s effectiveness, we define accuracy as the percentage of lines that T\textsc{inferrer} 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

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 intuitionally 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. 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\textsuperscript{nd} and 7\textsuperscript{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, T\textsc{inferrer} can be flexibly configured to favor any of the alternative template mappings for a translation task.

Template mappings ranked 3\textsuperscript{rd}, 4\textsuperscript{th}, 7\textsuperscript{th}, and 8\textsuperscript{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\textsuperscript{th} mapping when comparing two syntax trees, T\textsc{inferrer} 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\textsuperscript{th} and 8\textsuperscript{th} mappings, T\textsc{inferrer} can correctly translate `if (!m.aname()) to if (!(m.sname()))`, when there is an argument mapping `(p0, m.aname(), m.sname())` in its database.
We conducted three experiments to measure TINFERENCE’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 TINFERENCE’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 TINFERENCE’s cross-project translation (Section 4.4). Finally, in the third experiment, we compared the output produced by TINFERENCE 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 TINFERENCE’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

<table>
<thead>
<tr>
<th>Rank</th>
<th>Common Type</th>
<th>S</th>
<th>Swift template</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TypeDecl</td>
<td>73% (27/37)</td>
<td>100% (3/3)</td>
<td>0% (0/1)</td>
</tr>
<tr>
<td>2</td>
<td>MethodDecl</td>
<td>80% (34/42)</td>
<td>70% (7/10)</td>
<td>36% (4/11)</td>
</tr>
<tr>
<td>3</td>
<td>FieldDecl</td>
<td>75% (24/32)</td>
<td>88% (42/48)</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>Statement</td>
<td>83% (193/235)</td>
<td>92% (146/159)</td>
<td>36% (4/11)</td>
</tr>
<tr>
<td>5</td>
<td>Average</td>
<td>81% (246/346)</td>
<td>88% (229/260)</td>
<td>60% (15/25)</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>Rank</th>
<th>Java syntax type</th>
<th>Java template</th>
<th>Swift template</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>localVarDeclStmt</td>
<td>$p2 = p0$</td>
<td>var $p0 = $p1$</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>statement</td>
<td>if $p0$ {...}</td>
<td>if $p0$ {...}</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>importDecl</td>
<td>import $p0$</td>
<td>import $p0$</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>statement</td>
<td>$p0 = p1$</td>
<td>$p0 = p1$</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>classBodyDecl</td>
<td>public $p0$</td>
<td>public var $p1$ : $p0$</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>statement</td>
<td>return $p0$</td>
<td>return $p0$</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>statement</td>
<td>if $p0$ {...}</td>
<td>if $p0$ {...}</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>expression</td>
<td>! $p0$</td>
<td>! $p0$</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>classBodyDeclStmt</td>
<td>private $p2$ $p0 = p1$</td>
<td>private var $p0 = $p1$</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>classBodyDecl</td>
<td>public $p0$($p2 $p3)</td>
<td>public func $p1($p3: $p2) -&gt; $p0 {...}</td>
<td>19</td>
</tr>
</tbody>
</table>
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 j2swift’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
\[
\text{if(set.getEntryCount()} \geq \text{max.getEntryCount()} \text{)} \text{then}
\]

Without encoding the domain knowledge of mapping member APIs (e.g., getEntryCount() vs. 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.

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, 1710765, 1650540 and 1565827.

REFERENCES


