Facilitating the Development of Cross-Platform Software via Automated Code Synthesis from Web-Based Programming Resources[☆]

Sanchit Chadha[a,*], Antuan Byalik[a], Eli Tilevich[a], Alla Rozovskaya[b]

[a]*Software Innovations Lab*
Virginia Tech, Blacksburg, VA 24061, USA
[b]*Department of Computer Science*
Virginia Tech, Blacksburg, VA 24061, USA

Abstract

When a mobile application is supported on multiple major platforms, its market penetration is maximized. Such cross-platform native applications essentially deliver the same core functionality, albeit within the conventions of each supported platform. Maintaining and evolving a cross-platform native application is tedious and error-prone, as each modification requires replicating the changes for each of the application's platform-specific variants. Syntax-directed source-to-source translation proves inadequate to alleviate the problem, as native API access is always domain-specific.

In this article, we present a novel approach—*Native-2-Native*—that uses program transformations performed on one platform to automatically synthesize equivalent code blocks to be used on another platform. When a programmer modifies the source version of an application, the changes are captured. Based on the changes, *Native-2-Native* identifies the semantic content of the source code block and formulates an appropriate query to search for the equivalent target code block using popular web-based programming resources. The discov-

[☆]This article is an extended and revised version of a previous paper—"Native-2-Native: automated cross-platform code synthesis from web-based programming resources," published in Proceedings of the 14th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015), http://dx.doi.org/10.1145/2814204.2814210.

[*]Corresponding authors.

Email addresses: `schadha@cs.vt.edu` (Sanchit Chadha), `antuanb@cs.vt.edu` (Antuan Byalik), `tilevich@cs.vt.edu` (Eli Tilevich), `alla@vt.edu` (Alla Rozovskaya)

ered target code block is then presented to the programmer as an automatically synthesized target language source file for further fine-tuning and subsequent integration into the mobile application's target version. We evaluate the proposed method using common native resources, such as sensors, network access, and canonical data structures. We show that our approach can correctly synthesize more than 74% of iOS code from the provided Android source code and 91% of Android code from the provided iOS source code. The presented approach effectively automates the process of extracting the source code block's semantics and discovering existing target examples with the equivalent functionality, thus alleviating some of the most laborious and intellectually tiresome programming tasks in modern mobile development.

Keywords: Recommendation Systems, Code Synthesis, Mobile Computing, Android, iOS, Java, Swift

1. Introduction

Major mobile platforms, including Android, iOS, and Windows Phone, vigorously compete to dominate market share. The resulting mobile market fragmentation complicates the software process of those software vendors that strive to maximize their customer base: each mobile application has to be supported on all major platforms. Factoring out the conventions and formats of individual platforms, application replicas essentially deliver an identical set of functionalities. For example, a mobile application with a map component would use Google Maps on Android and Apple Maps on iOS Devices. Even though from the end-user's perspective, the map feature provides identical functionality on both platforms, from a software engineering perspective, implementing the same feature on different platforms requires the use of vastly dissimilar languages, APIs, and software architectures. For instance, Android applications are written in Java using the Android standard library, in which UI events are expressed by means of callbacks; meanwhile, iOS applications are written in Swift using the iOS standard library, in which UI events are expressed by means of delegates.

As new features and functionalities are being added to a mobile application, its developers must replicate the changes on all supported platforms. Having invested the time and effort required to ascertain the application logic and implementation details on one platform, the developer then essentially repeats the same process on all the remaining supported platforms. That is, the expertise gained by undertaking a programming task on one platform does not translate into facilitating the performance of essentially the same task on other platforms. What if it were possible to automatically glean the knowledge acquired by adding a feature to an application on a source platform to semi-automatically synthesize the code required to add the same feature on target platforms?

In this article, we present `native-2-native`—a novel approach that develops a code synthesis algorithm to discover publicly available target code blocks whose semantics are equivalent to a code block written for the source platform. From a high-level perspective, the `native-2-native` approach works as follows. When a programmer adds a feature to an application, the manually written code is logged. A query is created from the code to search the web for the available target platform code that implements the same functionality. The resulting web pages are ranked using a ranking algorithm that applies a high-dimensional feature vector to select the code block whose functionality is the closest to the original code. The selected code then parameterizes a code generator that synthesizes a semantically correct source code file that can be included into the target application's codebase—thus requiring minimal manual fine-tuning in most cases.

It is the ability to automatically discover a code block that natively implements a feature in Swift/iOS for an equivalent code block implemented natively in Java/Android, and vice versa, that gives our approach the name of `native-2-native`. Our approach focuses on the native resources in mobile applications, such as sensors and services. Because the majority of mobile applications nowadays need to make use of such native resources, our approach aims at facilitating one of the most common programming tasks undertaken by the modern mobile application developer. Hence, our approach would be most applicable to adaptive maintenance, enhancing existing applications with new features [1], as well as new development.

This article revises and expands our earlier paper [2] presented at the 14th International Conference on Generative Programming: Concepts & Experience (GPCE'15). For the journal edition, we extended our approach for bi-directional operation. That is, while the original version works only from Android/Java to iOS/Swift, the current version works also from iOS/Swift to Android/Java. This major extension required us to essentially double our evaluation to assess the effectiveness of our approach in both directions. Our revised and enhanced evaluation required us to refine and expand our experiments with new and updated analysis results now presented for both directions.

Our reference implementation of `native-2-native` extends the Eclipse IDE with a plugin to support cross-platform mobile development. The plugin captures and analyzes the token frequency in the Java or Swift code block that utilizes or accesses a resource by means of some native API. Based on the captured code, the plugin forms a query to search popular web-based programming resources for Swift or Java code blocks accessing equivalent native APIs on the iOS or Android platforms. The highest ranked discovered Swift/Android code blocks are provided to the developer, who can further refine them with extra functionality, such as fault tolerance capabilities or strengthened security.

We conduct an evaluation of test cases that include multiple application methods, converting code blocks from Android to iOS and in the reverse direction. We show that the proposed approach provides a valuable tool that can be used by developers charged with the challenges of supporting mobile applications on multiple platforms. We also show that the effectiveness of our method depends on the source and the target platforms. More specifically, evaluating the fitness of automatically discovered code for accessing subject native APIs for the task at hand showed that the approach is effective in 74% of test cases when going from Android to iOS and in 91% of test cases when going in the opposite direction.

The rest of this article is organized as follows. Section 2 presents a running example. Section 3 describes the approach. In Section 4, we present experimental results, and in Section 5, we discuss strengths and limitations of the proposed method. Section 6 discusses related work. Section 7 presents future work directions and conclusions.

```
public static LocationModel getLocation(Context context) {
    locationManager = (LocationManager)context.
      getSystemService(Context.LOCATION_SERVICE);
    Criteria criteria = new Criteria();
    String bestLocation = locationManager
      .getBestProvider(criteria, false);
    Location location = locationManager.
      getLastKnownLocation(bestLocation);
    LocationListener loc_listener = new LocationListener()
    {
        public void onLocationChanged(Location l) {}
        public void onProviderEnabled(String p) {}
        public void onProviderDisabled(String p) {}
        public void onStatusChanged(String p, int status,
          Bundle extras) {}
    };
    locationManager.requestLocationUpdates
      (bestLocation,0,0,loc_listener);
    location = locationManager.
      getLastKnownLocation(bestLocation);
    LocationModel loc = new LocationModel
      (location.getLatitude(),location.getLongitude());
    return loc;
}
```

Figure 1: Android get location basic functionality

2. Running Example

We provide a concrete example to motivate the need for `native-2-native`. Consider a mobile application that determines if any of the user's friends are

```
func startLocationUpdate() {
    locManager.requestWhenInUseAuthorization()
    locManager.startUpdatingLocation()
}

func locationManager(manager: CLLocationManager!,
  didUpdateLocations locations: [AnyObject]!) {

    var location = locations.last as! CLLocation
    var lat:Double = location.coordinate.latitude as
      Double
    var long:Double = location.coordinate.longitude
      as Double
    var result = NSString(format: "%.5f, %.5f",
      location.coordinate.latitude,
      location.coordinate.longitude)
    self.location = result as String;
}

func locationManager(manager: CLLocationManager!,
  didFailWithError error: NSError!) {

    DLog("Location Error: " + error.description);
}
```

Figure 2: iOS get location basic functionality

in the vicinity. Hence, the application is in essence a person proximity locator that continuously retrieves and processes GPS location information from the requesting device. Let us assume that the application is supported on both Android and iOS.

2.1. Obtaining GPS Location Information

Figure 1 shows the code block that retrieves GPS location in Android; Figure 2 shows equivalent functionality in iOS. Even though both code blocks accomplish the same task, they are structured quite dissimilarly. In particular, the

Android version creates a `locationManager` object from the passed context object, so that the initialized object can be queried for the GPS information. The iOS version creates a `CLLocationManager` object, whose initialized state contains the latitude and longitude variables. The code blocks on both platforms are relatively short, following similar coding idioms (i.e., creating an object to query its state) to retrieve the GPS information. Nevertheless, having written the code block in Android would not equip the programmer with the required knowledge to replicate this basic functionality in Swift.

Despite the popularity of source-to-source translators, they would be inapplicable if one wanted to automatically derive the iOS version of the code. The reason for the ineffectiveness of source-to-source translation in this instance is that native API access is always domain-specific, a complication that cannot be tamed with syntax-directed translation. By contrast, `native-2-native` attempts to identify the semantic content of the source code block and formulates an appropriate query to search for the equivalent target code block. By automating the process of extracting the semantics of the source code block and by discovering existing target examples with the equivalent functionality, the presented approach can alleviate some of the most laborious and intellectually tiresome programming tasks in modern mobile development.

2.2. Enabling Insights

Heretofore, the discussion in this article has focused on the *what* component of our approach—our end goal of automatically synthesizing native code for a target platform from equivalent code on a source platform, thereby enabling a cross-platform translation of natively implemented features. Before discussing the *how* component, i.e. the algorithmic and implementation details, we will address the *why* component, which will identify enabling insights of our approach.

The presented approach is enabled by a confluence of the following insights, derived from observing the realities of modern mobile software development: the peculiarities of the mobile software market, the working preferences of the modern mobile programmer, and the nature of platform-specific mobile APIs.

We next describe each of these insights in turn.

Major mobile platforms have been competing with each other for market domination. Mobile hardware vendors have embraced the competitive mindset, which results in a so-called “arms race” when it comes to the devices’ features and capabilities. As soon as one platform introduces a new hardware enhancement, the competitors feel compelled to introduce the equivalent or improved enhancement on their platform, lest they were to lose a major marketing advantage. Consider the GPS sensor, which provides the foundation for our running example. If this sensor and its corresponding capabilities were to be introduced to the Android platform first, the iOS platform would have not only to mimic this feature, but also to add extra enhancements in the closest release feasible. Android, in its turn, would be compelled to match the latest iOS progress in this area. This continuous competitive cycle, although driven exclusively by market forces, unveils the first enabling insight of our approach: major mobile platforms necessarily share an excessive amount of core native features.

Although the implementation languages and the corresponding native APIs of the major mobile platforms typically differ, the underlying feature sets from the end user’s perspective enjoy a remarkable degree of similarity, which, in turn, leads to a high level of correlation in the vocabularies used to express the native APIs for these features. Assuming that API designers aim at creating intuitive-sounding and easy-to-understand names, reading in GPS information, for example, can be expressed in a finite, reasonably sized number of ways. Moreover, one can expect that the ways the GPS APIs are expressed on different mobile platforms will overlap in non-insignificant ways. Going back to the code blocks for this feature in Figures 1 and 2, one can see that the tokens *location* and *location manager* are both heavily used in both Java/Android and Swift/iOS. It is these shared vocabularies that make it possible to design a cross-platform translation mechanism for native APIs for shared features. Furthermore, the number of analogous features and their corresponding API shared tokens will continue to increase as long as there is competition among the major platforms on the market.

Based on the growing number of online programming resources, the modern programmer increasingly relies on the Web to answer questions that come up during their day-to-day operations ranging from simple bug fixes to complex refactoring. For example, StackOverflow [3] receives 2.65 new questions per minute and 4.41 new answers per minute on average, reaching close to 6,000 questions a day in peak sessions [4]. Indeed, StackOverflow and similar online question/answer discussion forums have become the modern programmer's primary medium of information exchange. One can attribute this strong shift toward online programming documentation to sheer demographics—StackOverflow reports that the average age of their user is 28.9 year old, based on surveying over 26K developers across 157 countries [5], which places them strongly within Generation Y, also known as the first digital generation, used to rely on the Web for all kinds of information. At any rate, it is indisputable that the Web has become an invaluable information sharing and acquisition platform for mobile developers. An additional draw of programming resources web sites is serving as reputation builders. Users of these websites commonly have the ability to rank the quality of provided information, with top providers earning high degrees of prestige and notoriety. These digital rankings can also be leveraged to automatically assess the quality of the available programming information.

Finally, we argue that figuring out how to express a native API on some platform, having just expressed an equivalent API on another platform, constitutes *accidental complexity*, and as such is a promising candidate for automated treatment via software engineering innovation [6]. Consider the process by which some native API becomes used in a typical mobile program. A developer decides that some feature needs to be added to a program, and that feature will make use of some native resource. Designing the feature is *essentially* complex, while discovering which API one must invoke is *accidentally* complex. Furthermore, this discovery process remains accidentally complex, irrespective of the implementation platform. Removing accidental complexity is a key driving force behind reducing the programmer workload. There is another peculiarity that arises when translating native APIs between Java and Swift. While Java remains the most commonly used programming language [5, 7], Swift according to StackOverflow is now regarded as "most loved" language. Hence, one can expect an abundance of web-based programming resources for both languages.

By leveraging these three main insights, we were able to create a simple but powerful approach to automatically translate native APIs between Java/Android and Swift/iOS and vice versa. In the next section, we provide the algorithmic and implementation details of our approach. In Section 4, we report on the results of applying our approach to real-world examples of using native APIs.

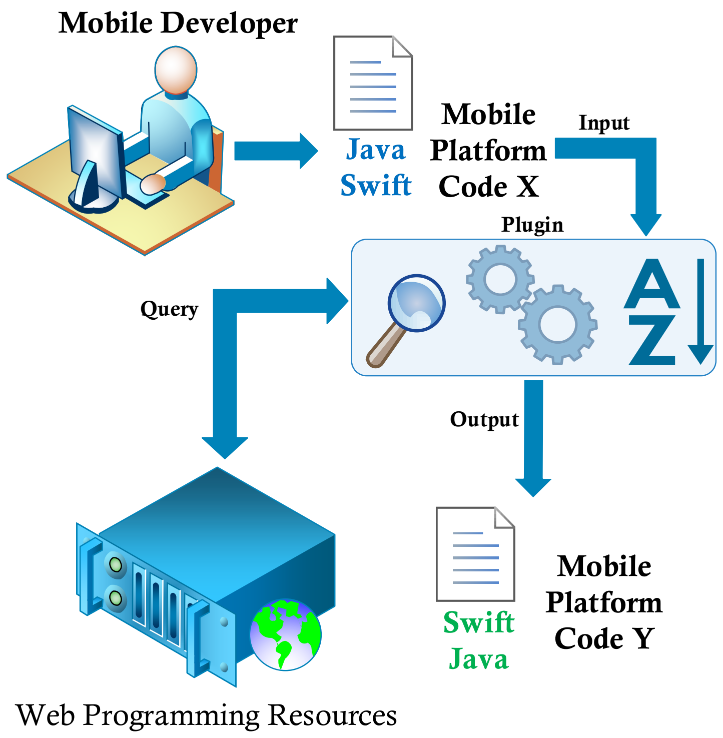


Figure 3: Native-2-Native: High-level Approach Overview

3. The Native-2-Native Approach

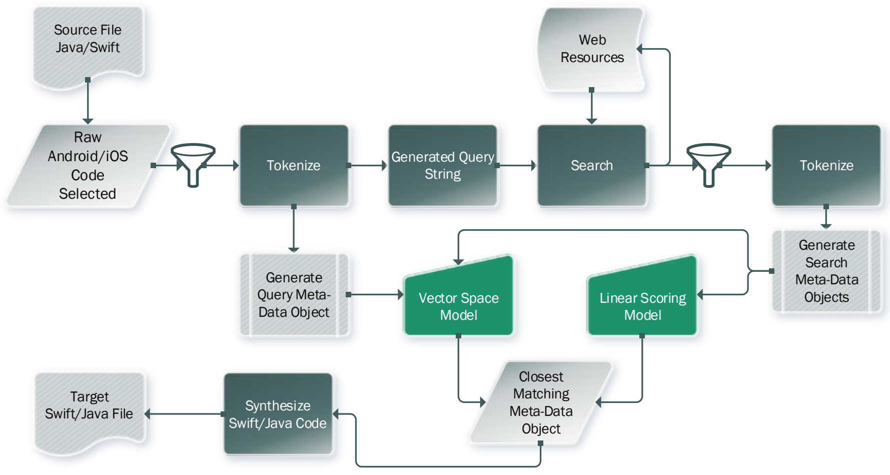

Source File Java/Swift
Raw Android/iOS Code Selected
Tokenize
Generated Query String
Web Resources
Search
Tokenize
Generate Search Meta-Data Objects
Generate Query Meta-Data Object
Vector Space Model
Linear Scoring Model
Closest Matching Meta-Data Object
Target Swift/Java File
Synthesize Swift/Java Code

Figure 4: Full approach program flow

Figure 3 shows a high-level overview of our approach. A mobile application developer first implements a new feature using some native API of the Android platform in Java or of the iOS platform in Swift. Once the developer deems the feature's implementation completed, the feature's code block is passed as input into the `Native-2-Native` IDE plugin. The plugin performs the following tasks in sequence: generalize the Java/Swift code block to form a search query, execute the query against popular web-based programming resources, summarize and rank the results, and, finally, present a synthesized Swift code block for the iOS platform or Java code block for the Android platform back to the developer. The presented code block is typically partially complete and implements the same feature as the input code, but by means of the equivalent native target platform's API.

In the following sections, we detail the constituent parts of the native-2-native approach. Section 3.2 describes the process of extracting core functionality from the input Java/Swift code block. This process includes multiple techniques in tokenization, filtering, and frequency analysis to generate an important set of query keywords. Next, Section 3.3 explains how the approach ascertains a similarity index between the initial Java/Swift code block and all mined Swift/-Java code blocks. The meta-data objects serve as the central representation of platform and language-independent semantics for all mined documents as well as the input source code. Finally, Section 3.4 delves into the underlying process by which the approach is able to produce an equivalent target platform code block. This process makes use of two algorithms: (1) a searching algorithm that discovers the relevant resulting set of documents, and (2) a ranking algorithm that operates on a set of mined documents to produce a platform- and language-independent semantic ranking, which synthesizes the output code block.

3.1. Terminology

For the rest of this article, the term *source document* will refer to the input code block (written in Java or Swift), while *target document* will refer to the output code block (synthesized in Swift or Java, respectively). *Query keywords* will refer to terms extracted from the input code block that describe its functionality and that are used to search for the target document's constituent components. Finally, *token* will refer to any single document element at the level of individual space-separated strings.

3.2. Extracting Core Functionality from Source Input Code

This section describes the process of translating input code blocks into web queries. This process extracts the core functionality of the code block by means of tokenization, filtering, and frequency analysis.

3.2.1. Tokenizing and Filtering

The flowchart in Figure 4 details the process. The Java/Swift source input is tokenized, with the superfluous tokens filtered out. As a result, a unique list of tokens in the source document is generated. The tokenization procedure makes use of the canonical bag-of-words model [8]. This model also separates camel case variables, title case variables, method declarations/invocations, and also removes non-alphanumeric characters attached to strings.

The resulting tokens are then normalized by applying stemming. Specifically, we remove verb inflections to eliminate verb-tense discrepancies that could arise while searching web resources for the target document's constituent components. Then tokens that happen to be substrings of other tokens are filtered as well. For example, tokens 'location' and 'loc' are assumed to possess related semantic intent. Tokens that are likely to weaken the precision of the frequency analysis are filtered out as well (e.g., comment designators, stop words, etc.)

3.2.2. Frequency Analysis to Generate Web Queries

The next step calculates the document frequency for each unique term occurring in the source document that has not been filtered. The frequency analysis produces a sorted list of the most used unique terms in the input. The top k terms become the query keywords for the search routine in Figure 5. The default value of k used in this work is 3 but can be customized at will. These top 3 terms are then used to produce a set of queries as follows: for every subset of terms from the list, we generate all possible term permutations, so that each query contains up to 3 terms. The retrieved webpages are the union of all results returned by each of the queries.

The procedure above uses a fixed number of terms to produce a query and, while it uses Information Retrieval methods, it can be further enhanced via Natural Language Processing techniques. Identifying most informative terms to formulate a query is an important component of the proposed approach, but is outside the scope of this work and we leave it for future research.

3.3. Meta-Data Objects

Meta-data, data that describes other data, has been used to facilitate the search and discovery of related data objects [9, 10]. The presented approach uses meta-data to describe source and target documents as a means of streamlining similarity comparison. The source document's meta-data object is composed upon the completion of the tokenizing and filtering steps as described next.

3.3.1. Meta-Data Fields

Meta-data fields serve as the abstraction that captures all pertinent information from web-based programming resources and the source document in an easily search-able and comparable format. The meta-data objects use their fields to store features mined from all the searched web resources as well as those extracted from the source document. These fields also include general information about each document that help identify categories used in later routines. This differs from a pure feature vector which only contains the necessary features for use in searching and ranking. A `generate` routine processes every search result to populate the fields defined by a given meta-data object. Representing the search results via meta-data objects makes them easily amenable to similarity evaluation with various ranking models.

Some features are deliberately supplemented to indicate the functionality to search for. For example, a preferred StackOverflow response would be a so-called "accepted answer," a code snippet check-marked by its originator as having solved the posed question.[1] In contrast, the source document lacks this property, as it simply represents the input Android or iOS code block. To prevent supplemented features from unduly skewing the final ranking, the source document's missing features are defaulted in the meta-data object as their ideal functionality. Similarly, web search results returned from random programming resources will also contain missing features, as well their own supplemental features, to be steered toward the searched for functionality. The meta-data objects' fields are the union of all features across all mined web resources and the source document.

[1]The StackOverflow website uses green check-marks to denote accepted answers.

3.4. Searching and Ranking

In this section, we first present a searching algorithm that queries web-based programming resources for relevant data pertaining to the query keywords. Then, we present a ranking algorithm that incorporates two different models for determining similarity between the source document and all mined potential target documents.

It is worth pointing out that both the searching and ranking algorithms disregard the control flow constructs present in the source document, operating solely on the extracted keywords described in the previous section. This design decision renders our approach largely independent of the specifics of the business logic of the native code blocks at hand, focusing exclusively on the native API used. Nevertheless, the disregarded control flow constructs are fully restored during the final code synthesis phase.

3.4.1. Searching Algorithm

Figure 5 shows the searching algorithm for mining the relevant data. The algorithm's input is the generated query keywords discussed above and the output is a full list of the resulting searches stored in a custom-made, answer-wrapper object. Before the core of the searching algorithm is executed, the `initKeywords` subroutine first initializes the full query keyword set. Although the input is just the set of query keywords, various permutations and subsets of the original query keywords must be searched to locate all relevant results. The `initKeywords` subroutine in Figure 5 explains how the full set of keywords involves three components: 1) transcribe the original keywords as individual queries, 2) identify the first and second (by frequency-based importance) keywords, in both orders, as subset queries, and 3) permute the complete set of the original keywords. Although some flexibility in the number of query keywords is allowed for user fine-tuning, the hard limit of 5 separate keywords for the permutation component ensures that the input is computable in practical space and time boundaries.

```
/* INPUT: query keywords to search */
/* OUTPUT: full list of resulting searches */
DEF Search($keywords$)
    $resultList \leftarrow \emptyset$
    $keywordSet \leftarrow$ initKeywords($keywords$)
    FOREACH $keyword \in \forall keywordSet$ DO
        $queryResultsSOF$ = execStackOverflow($keyword$)
        FOREACH $result \in \forall queryResultsSOF$ DO
            $Ans \leftarrow result, rank$
            $resultList \leftarrow resultList \cup$ {Ans}
        END FOREACH

        $queryResultsGoogle$ = execGoogle($keyword$)
        FOREACH $result \in \forall queryResultsGoogle$ DO
            $temp \leftarrow$ execStackOverflow($result$)
            $Ans \leftarrow temp, rank$
            $resultList \leftarrow resultList \cup$ {Ans}
        END FOREACH

        $queryResultsElse$ = execElse($keyword$)
        FOREACH $result \in \forall queryResultsElse$ DO
            $Ans \leftarrow result, rank$
            $resultList \leftarrow resultList \cup$ {Ans}
        END FOREACH
    END FOREACH
    RETURN $resultList$
END Search

DEF initKeywords(keywords)
    fullSet $\leftarrow$ keywords
    fullSet $\leftarrow$ fullSet $\cup$ {key[0] + key[1]}
    fullSet $\leftarrow$ fullSet $\cup$ {key[1] + key[0]}
    fullSet $\leftarrow$ fullSet $\cup$ perm($keywords$)
    RETURN $fullSet$
END initKeywords
```

Figure 5: Search Algorithm Pseudo code

Although the algorithm can be configured to search any web-based programming resources, we next describe it by means of three representative categories: (1) a live API call to StackOverflow with the given query keywords, (2) a Google search on the current query keywords that specifically limits to StackOverflow websites only to catch discrepancies in a StackOverflow search internally as well as an external Google search on the same keywords; the Google search results are funneled to StackOverflow to directly data-dump that post.[2] (3) a Google search that excludes StackOverflow posts to include less popular but potentially also relevant online blogs and other web resources to the list of relevant search results. All three methods' results are standardized into the answer-wrapper object and stored for later ranking and analysis. This entire process is conducted for each of the permuted list of query keywords to generate the final list of web-based search results.

3.4.2. Ranking Algorithm

The ranking algorithm determines the degree of similarity between the source document and all potential target documents to present the most relevant Swift or Android code blocks (depending on the input code block) to the user. Figure 6 details the algorithm. The input is the list of search results generated by the searching algorithm, as well as the original source document (also in the answer-wrapper format), and a k value for the number of results to be returned. The output is the k-top results of the ranking. The algorithm's main components produce the vector space model and linear model with their respective feature sets. The standardization into answer-wrapper objects described above facilitates the retrieval of this information for each document in the results list. The standard format for the answer-wrapper objects is used to generate the target set of meta-data objects that produce their respective vector space and linear model scores. Next, the models are combined, sorted, and the top k of those

[2]Searching through Google and StackOverflow frequently yields dissimilar complementary results in differing orders.

are returned.

Figure 6 also includes the subroutine for calculating the `cosine` of the angle between the current potential target document and the source document, as required by the central logic of the vector space model [11]. The cosine determines similarity from a constructed n-dimensional sphere, containing all the n-dimensional feature vectors. Each vector's dimension is calculated using a term-frequency inverse-document-frequency (`tf-idf`) weighting function, which accounts for a term's frequency without over fitting by accounting for common terms across the document corpus, where n is the total number of unique tokens available in the current document corpus. Each value in the vector is represented by the modified indicator function seen in equation 1 that incorporates the tf-idf weights. Here $d \in D$ represents some document in the full k set of documents, including all potential targets and the source, with weight w.

$$v_{d_k}(i) = \begin{cases} 0 & \text{if token } i \notin d_k \\ w_i & \text{if token } i \in d_k \end{cases} \tag{1}$$

The cosine value is calculated using the underlying vector space model shown in Equation 2 [10]. The norm of each vector is the square root of the summation of each dimension's squared value. Given that the dot product of two vectors is a scalar and this quantity is divided by the product of two norms, the resulting value is also a scalar. The variables s and t are used to denote the source and target documents, respectively.

$$cos_s(t) = \frac{s \cdot t}{||s||_2 \quad ||t||_2} \tag{2}$$

Note that the linear model subroutine is not shown in figure 6, as it is simply the linear combination of all relevant features in each meta-data object of the results documents shown in equation 3. The model's weights are derived from a combination of tf-idf values and normalization by average feature quantities

```
/* INPUT: source/targets MetaData, k top results */
/* OUTPUT: k closest ranked MetaDatas */
DEF Rank($Source, TargetMDs, k$)
    $ranking, topK \leftarrow \emptyset$
    FOREACH $target \in \forall TargetMDs$ DO
        $C \leftarrow$ getCos($target, source$)
        $L \leftarrow$ getLin($target$)
        $temp \leftarrow$ bag($C, L$)
        $ranking \leftarrow ranking \cup$ (temp, target)
    END FOREACH
    sort($ranking$)
    FOR $i$ IN $k$ DO
        $topK \leftarrow topK \cup ranking(i)$
    END FOR
    RETURN $topK$
END Search

DEF getSimilarityUsingCos($t, s$)
    $N_t, N_s, cos \leftarrow \emptyset$
    FOREACH $d \in \forall t$ DO
        $N_t \leftarrow N_t, d^2$
    END FOREACH
    FOREACH $d \in \forall s$ DO
        $N_s \leftarrow N_s, d^2$
    END FOREACH
    $N_t \leftarrow sqrt(N_t)$
    $N_s \leftarrow sqrt(N_s)$
    $cos \leftarrow \frac{t \cdot s}{N_t \cdot N_s}$
    RETURN $cos$
END getCos
```

Figure 6: Rank Algorithm Pseudo code

available from the StackOverflow API.

$$\text{lin}(t) = \sum_{i=1}^{n}(w_i * t_i) \tag{3}$$

3.4.3. Complexity Analysis

In this section, we briefly comment on the computational complexity incurred by the more exhaustive procedures of the `native-2-native` approach. The ranking algorithm and following model combination process is the most computationally intensive component, on which we hence concentrate our analysis. Recall Equations 1 and 2, presented in the previous section, that outline the vector space model approach, as well as Equation 3 that explains the linear model used. Equation 4 shows the complete model combination, along with the weight calculation, as a function on all potential target documents. Note the parameters α and β that are constrained such that $\alpha + \beta = 1,\ \alpha \geq 0,\ \beta \geq 0$ and represent the respective magnitude of the two models to each other.

$$\max_{t \in T} \left\{ \frac{\alpha(s \cdot t_j)}{\sqrt{\sum_{i=1}^{n} s_i^2 \cdot \sum_{i=1}^{n} t_{i,j}^2}} + \frac{\beta \sum_{i=1}^{k} w_{i,j} f_{i,j}}{\max\limits_{a \in A} \left(\sum_{i=1}^{k} w_{i,a} f_{i,a} \right)} \right\} \tag{4}$$

This equation is derived by combining the vector space model with a normalized linear model and using the weighting function shown in Equation 5. In Equation 5, note the $I(t)$ indicator function that is 0 if the current term is not present in the current document and 1 otherwise. Also of importance to this equation is T which represents the total number of target documents mined by the searching algorithm.

$$s, \forall t_j \in T : t_f \cdot \log\left(\frac{|T| + 1}{I(t)}\right) \tag{5}$$

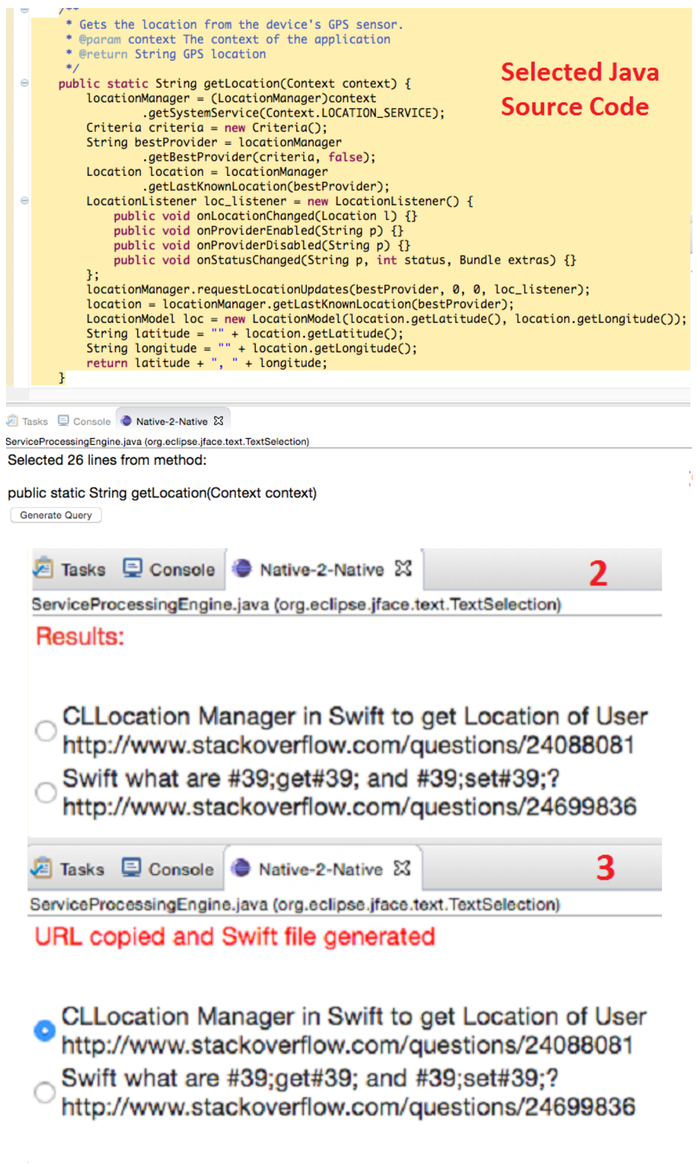
Given that n_i is the total number of tokens in some document i, we bound the maximum number of tokens on the overall approach as $dim = \{\sum_{i=1}^{|T|+1}\{n_i\}\}$. This bound being placed in set notation to represent the elimination of duplicate tokens across not just a single document but all documents, and accounting for the full set T of target documents in addition to the single source document.

Lastly, d is the upper bound on the feature vector used for the vector space model as we take all unique tokens. In essence, we have $|{<}d_{1,i}, d_{2,i}, ..., d_{n,i}{>}| \leq dim$, where the vector represents some document d_i's n features; that will be equal across all targets and the single source document. Hence, the Big O of the approach's primary component in terms of documents is $\mathcal{O}(T + 1)$, which runs in linear time, $\mathcal{O}(n)$. However, accounting only for documents fails to accurately reflect the true computational complexity, as for certain operations one must measure their more-numerous token operations to evaluate their complexity. Thus, the complexity measured in tokens is $\mathcal{O}(dim^2 + dim * d_i + dim * f + f)$, where f is the number of linear features, a number strictly smaller than dim, and where d_i is the current document's token count, also strictly smaller than dim as shown above. If $d_i \leq dim$, then $d_i * dim \leq dim^2$. Finally, we can simplify the overall complexity in terms of tokens as $\mathcal{O}(dim^2 + dim^2 + dim^2 + f) = \mathcal{O}(3dim^2) = \mathcal{O}(n^2)$.

3.5. Programming Interface and Code Synthesis

The approach is concretely realized as an Eclipse IDE plugin publicly available and open-sourced for future improvements and enhancements at `https://github.com/antuanb/Native-2-Native`. Figure 7 (top half) displays the initial plug-in view from an Android mobile developer's perspective. The developer first highlights a code block to be rendered in Swift (since the source file is in Java for Android) and then clicks the generate button. Figure 7 (bottom half) highlights how the developer selects which of the top two presented results they desire to be presented as a Swift source file. Note the URL of the corresponding result is copied to the clipboard, so that the developer can refer to the originating web-resource for further information. The generated Swift source file is saved in the current working directory of the Android/Java project.

The top half of Figure 8 shows a sample generated Swift file in our running example of GPS location. The control flow of the Android/Java file is replicated to create a skeleton Swift source file first. Then, the developer-selected result is incorporated into the Swift file to finalize the synthesized code block. In



```
/**
 * Gets the location from the device's GPS sensor.
 * @param context The context of the application
 * @return String GPS location
 */
public static String getLocation(Context context) {
    locationManager = (LocationManager)context
            .getSystemService(Context.LOCATION_SERVICE);
    Criteria criteria = new Criteria();
    String bestProvider = locationManager
            .getBestProvider(criteria, false);
    Location location = locationManager
            .getLastKnownLocation(bestProvider);
    LocationListener loc_listener = new LocationListener() {
        public void onLocationChanged(Location l) {}
        public void onProviderEnabled(String p) {}
        public void onProviderDisabled(String p) {}
        public void onStatusChanged(String p, int status, Bundle extras) {}
    };
    locationManager.requestLocationUpdates(bestProvider, 0, 0, loc_listener);
    location = locationManager.getLastKnownLocation(bestProvider);
    LocationModel loc = new LocationModel(location.getLatitude(), location.getLongitude());
    String latitude = "" + location.getLatitude();
    String longitude = "" + location.getLongitude();
    return latitude + ", " + longitude;
}
```

ServiceProcessingEngine.java (org.eclipse.jface.text.TextSelection)

Selected 26 lines from method:

public static String getLocation(Context context)

ServiceProcessingEngine.java (org.eclipse.jface.text.TextSelection)

Results:

CLLocation Manager in Swift to get Location of User
http://www.stackoverflow.com/questions/24088081

Swift what are #39;get#39; and #39;set#39;?
http://www.stackoverflow.com/questions/24699836

ServiceProcessingEngine.java (org.eclipse.jface.text.TextSelection)

URL copied and Swift file generated

[✓ CLLocation Manager in Swift to get Location of User
http://www.stackoverflow.com/questions/24088081]

Swift what are #39;get#39; and #39;set#39;?
http://www.stackoverflow.com/questions/24699836

Figure 7: Plug-in showing user selecting GPS Location code and subsequent results returned

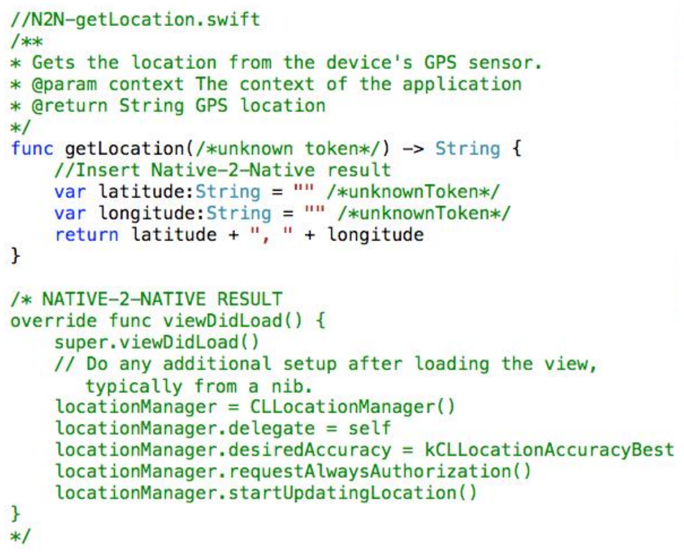
this example, the synthesized Swift file has the correct iOS/Swift protocol for instantiating and utilizing the `CLLocationManager` native API to access a user's GPS location along with a logic flow outline. The remaining fine-tuning left for the developer is to format and return the contained location information in the style desired.

3.6. Extension

Using the existing query extraction, searching and ranking algorithms of `Native-2-Native`, we successfully implemented a robust converse implementation that took Swift/iOS source code as input and returned Java/Android code snippets from web resources. Several Swift keywords were filtered out and an automated code detection system was implemented for the algorithm to detect what source code language was used as input. Both Swift and Java have distinct method headers which allowed for a straightforward detection of whether the intended search query was to find Java snippets or to find Swift snippets from web resources.

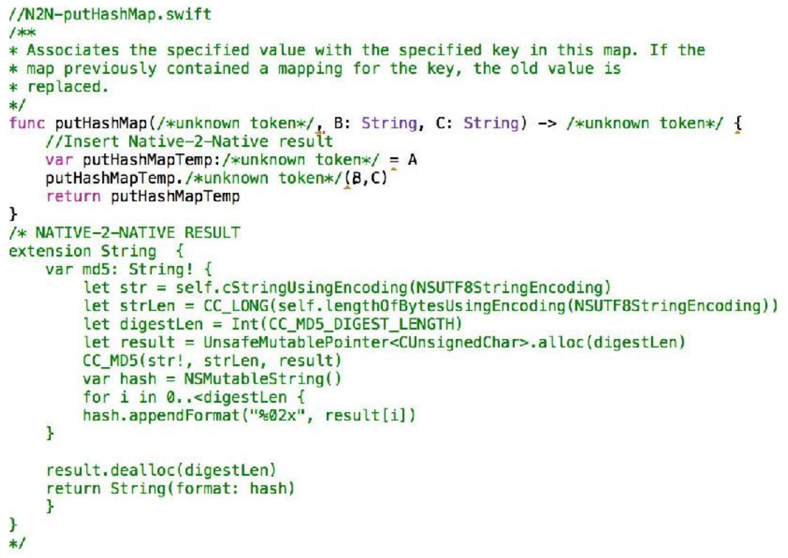
Swift is a much more declarative or concise language than Java and that made it slightly more difficult to generate relevant queries since there isn't as much code and context available for frequency analysis to be an accurate query keyword generator. On the other hand, Java is a much more mature language and therefore has a lot more solutions, code snippets and support available on web resources such as StackOverflow. This suggested that despite not having the best possible keywords in the query, there was still a higher chance of encountering a relevant search result and relying on our robust ranking algorithm to display that result in the top 1 or 2 positions for the programmer to use. This proposition was proven and discussed in Section 4, where tabular results are shown with analysis on what the numbers signify.

By implementing the bidirectional component, we pave the path for a more generalized code recommendation system which can be used for any combination of languages available. The next section focuses on the evaluation of the presented approach, detailing the native APIs used, the evaluation process, and the precision levels obtained.



```
//N2N-getLocation.swift
/**
* Gets the location from the device's GPS sensor.
* @param context The context of the application
* @return String GPS location
*/
func getLocation(/*unknown token*/) -> String {
    //Insert Native-2-Native result
    var latitude:String = "" /*unknownToken*/
    var longitude:String = "" /*unknownToken*/
    return latitude + ", " + longitude
}

/* NATIVE-2-NATIVE RESULT
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view,
       typically from a nib.
    locationManager = CLLocationManager()
    locationManager.delegate = self
    locationManager.desiredAccuracy = kCLLocationAccuracyBest
    locationManager.requestAlwaysAuthorization()
    locationManager.startUpdatingLocation()
}
*/
```



```
//N2N-putHashMap.swift
/**
* Associates the specified value with the specified key in this map. If the
* map previously contained a mapping for the key, the old value is
* replaced.
*/
func putHashMap(/*unknown token*/, B: String, C: String) -> /*unknown token*/ {
    //Insert Native-2-Native result
    var putHashMapTemp:/*unknown token*/ = A
    putHashMapTemp./*unknown token*/(B,C)
    return putHashMapTemp
}
/* NATIVE-2-NATIVE RESULT
extension String  {
    var md5: String! {
        let str = self.cStringUsingEncoding(NSUTF8StringEncoding)
        let strLen = CC_LONG(self.lengthOfBytesUsingEncoding(NSUTF8StringEncoding))
        let digestLen = Int(CC_MD5_DIGEST_LENGTH)
        let result = UnsafeMutablePointer<CUnsignedChar>.alloc(digestLen)
        CC_MD5(str!, strLen, result)
        var hash = NSMutableString()
        for i in 0..<digestLen {
        hash.appendFormat("%02x", result[i])
    }

    result.dealloc(digestLen)
    return String(format: hash)
    }
}
*/
```

Figure 8: Synthesized Swift code snippet for GPS Location (top – YES Result) & and HashMap (bottom – NO Result)

4. Evaluation

We now present the results of evaluating our approach. `Native-to-native` was evaluated using various APIs in Android/Java and iOS/Swift. The APIs

Table 1: Proof of concept - Android/Java → iOS/Swift

API Experiments	Total	Answer Exists	Pre-Rank 1	Pre-Rank 1 or 2	Rank 1	Rank 1 or 2
Services	6	2 (33%)	2 (33%)	2 (33%)	2 (33%)	2 (33%)
String	25	24 (96%)	23 (92%)	24 (96%)	21 (84%)	23 (92%)
ArrayList	22	17 (77%)	10 (45%)	12 (55%)	13 (59%)	14 (64%)
HashMap	13	13 (100%)	9 (69%)	10 (77%)	8 (62%)	10 (77%)
Totals	**66**	**56 (85%)**	**44 (67%)**	**48 (73%)**	**44 (67%)**	**49 (74%)**

Table 2: Proof of concept - iOS/Swift → Android/Java

API Experiments	Total	Answer Exists	Pre-Rank 1	Pre-Rank 1 or 2	Rank 1	Rank 1 or 2
Services	6	4 (67%)	2 (33%)	4 (67%)	2 (33%)	4 (67%)
String	25	24 (96%)	20 (80%)	23 (92%)	20 (80%)	24 (96%)
ArrayList	22	21 (95%)	11 (50%)	15 (68%)	19 (86%)	21 (95%)
Dictionary	13	12 (92%)	7 (54%)	7 (54%)	8 (62%)	11 (85%)
Totals	**66**	**61 (92%)**	**40 (61%)**	**49 (74%)**	**49 (74%)**	**60 (91%)**

included sensors (e.g. GPS, accelerometer), network interfaces (e.g. WiFi, Bluetooth Low Energy (BLE)), and canonical library classes/data structures (e.g. `String`, `ArrayList`, `HashMap`).

The main goal of the evaluation was to determine whether the approach proposed in this work is *suitable for adaptive maintenance* (i.e., adding new features to existing applications) and *new development*. The evaluation of the resulting synthesized functionality was performed by hand by the primary author, Sanchit Chadha. A graduate of Virginia Tech with 4 years of Java and Android experience, along with 2 years of iOS development experience with a majority of applications written in Swift, the primary author was the best candidate to evaluate the quality of the resulting code produced by `native-2-native`. Nevertheless, this reliance on a single evaluator presents an external threat to the validity of our experimental results, as we discuss in Section 5.

The manual annotation included compiling the obtained code with the target platform compiler and testing its runtime behavior. We evaluated both parts of the approach – *the query formulation component* and *the ranking algorithm* itself.

To evaluate the quality of the algorithm for generating queries, we checked for how many of the source code blocks a correct answer was present in at least one of the retrieved web pages. This evaluation also sets an upper bound on the performance of the ranking algorithm since the ranking algorithm is only able to succeed in those cases for which a correct answer exists among all the retrieved results.

For the evaluation of the ranking algorithm, the results were placed in one of the following categories: (1) **Yes**, the synthesized code is correct or salvageable for the implementation required or (2) **No**, the synthesized code is completely irrelevant to the input source code block and unsalvageable. Note that the above evaluation procedure is a modification of the evaluation presented in the earlier version of this paper [2] in two respects. First, we evaluate the contribution of each component; second, we eliminated the *Maybe* category when deciding whether the selected code block is relevant. The results are thus not directly comparable to those presented in the earlier version.

Tables 1 and 2 show the results of converting Java to Swift and Swift to Java, respectively. The `Answer Exists` column evaluates the query formulation algorithm by showing for how many test cases a correct answer exists in at least one retrieved result. This result is higher for the Swift $\rightarrow$ Java direction, 92% vs. 87% for the Java $\rightarrow$ Swift direction.

The `Pre-Rank 1` and `Pre-Rank 1 or 2` columns represent the correct answers that are in Rank 1 only or either in Rank 1 or 2, respectively, before our ranking algorithm is applied. This statistic serves as a baseline for the ranking algorithm, as it demonstrates the performance of the ranking algorithm in the commercial search engine that we query. As shown, pre-rank results for both directions are fairly similar, indicating the commercial ranking algorithm is performing consistently.

Finally, the last two columns, `Rank 1` and `Rank 1 or 2`, represent the correct answers that are in Rank 1 only or in either Rank 1 or 2, respectively, after our ranking algorithm is applied. If we consider the top 2 results after the application of the ranking algorithm, 74% of the Java to Swift direction and 91% of the Swift to Java direction are found relevant. In the second-to-last column (Rank 1) it is shown that 67% and 74% are relevant when the top 1 result only is considered. Compared to the baseline shown in columns `Pre-Rank 1` and `Pre-Rank 1 or 2`, we obtain a slight improvement of 1% when going from Java to Swift and a substantial improvement of 17% when going in the opposite direction. These improvements demonstrate that the features that our search algorithm is using and that are tailored to the task, are useful. The relevant features include the number of votes or number of views that an answer from a web resource has, as well as information about whether an answer has been officially accepted or not.

Overall, performance is better for the Swift $\rightarrow$ Java direction, showing how a more mature and imperative (wordy) language such as Java is more likely to produce relevant code snippets from web resources.

5. Discussion

The experimental results show that the approach is effective and can serve as a practical tool for mobile programmers who support cross-platform applications. Because the automatically suggested code does not need to be perfect to provide a high degree of utility to the programmer, our algorithms proved surprisingly fit for the purposes intended. However, it is not only the fitness of our algorithms that explains the effectiveness of our approach. These algorithms work hand-in-hand with the realities of the mobile market, the availability of web-based programming resources, and the process of suggesting equivalent native APIs being manageable via tool automation.

The development process of `native-2-native` is *automated* rather than *automatic*. That is, the code snippets returned by our web queries are not meant for immediate inclusion into the application codebase. The developer is expected to first examine and adapt them if necessary. Nevertheless, we expect that our approach can be conducive to boosting programmer productivity by automatically forming relevant search queries and ranking the results.

Despite its practical utility, the `native-2-native` approach has several limitations. The model underlying its searching and ranking algorithms is bound by the dimensions of the feature vector. In other words, the accuracy of the algorithms is inversely proportional to the size of the total number of unique tokens comprising a given code block. As a result, mobile developers are likely to find our approach most effective in those cases when they need to find equivalent iOS code for small to medium (10-30 lines of code) Android native code blocks. The second limitation stems from the original closed development model for the iOS platform. Closed models traditionally result in reduced sharing of programming solutions. Exacerbating the conditions for evaluating the applicability of our reference implementation is a relative newness of the Swift language. In fact, we were surprised that our reference implementation was able to synthesize correct suggestions from a relatively limited set of web-based Swift programming resources. Nevertheless, several major technological trends are likely to address this limitation. For one, Swift will be open-sourced in coming releases, while the amount of available Swift code examples on the web seems to grow by leaps and bounds.

Our experimental results are subject to both internal and external validity threats. The internal validity is threatened by our choice of the programming scenarios used in the experimental evaluation. Although selecting these particular scenarios is influenced by our own programming practices, the selections represent fairly standard examples of native API usage and canonical data structures, covering a wide range of APIs, both in Java and Swift. The external validity is threatened by relying exclusively on the judgment of the first author to evaluate the quality of the recommended code snippets. A mobile developer with a different technical background in terms of familiarity with the Android and iOS platforms may consider a different percentage of the recommended code snippets acceptable for further refinement and integration into the codebase. The first author's exposure to the Android and iOS platforms was limited to exclusively academic settings when the research was conducted, so seasoned mobile developers with industry experience may find a larger percentage of the recommended snippets useful. Additionally, using multiple annotators with dissimilar mobile experiences would likely yield additional insights and generalization. Hence, evaluating the `native-2-native` approach with respect to multiple developers remains an important direction for future work.

Finally, synthesizing Swift from given Java input is a necessarily difficult case of cross-language translation. Because Swift is much more declarative (i.e., concise) than Java, the translation must produce more declarative output from more loquacious input. As software engineering is becoming more declarative in terms of languages, specifications, and invariants, our approach holds a lot of promise for automatically transitioning current mainstream methods of expressing programming information into their declarative counterparts. For example, our approach can be used to get rid of the wordiness of anonymous inner classes in the pre-Java8 world, replacing this with code lambda expressions.

6. Related Work

`Native-2-native` is a representative of a broad class of software engineering applications known as recommendation systems [12, 13]. Several examples of recommendation systems synthesize code snippets from web-based programming resources [14, 15, 16, 17] or build an intelligent code search engine [18]. We will discuss how our approach differs from or improves over these examples.

Prompter [14] is an Eclipse plug-in that given the current working code context automatically identifies relevant StackOverflow discussions. Its uniqueness is in providing a user-controlled confidence threshold to suggest only discussions that surpass this threshold. Compared to `native-2-native`, Prompter also makes use of the StackOverflow API, albeit as the only source for relevant discussion and code snippets. A larger search space of our approach [19], which includes Google Search results in `native-2-native`, can achieve the level of precision required for cross-language and platform translation, a feature not supported by Prompter [14] or [17]. While Prompter locates and presents relevant code snippets, it does not synthesize new code like `native-2-native`, a feature that can make a major difference when it comes to user satisfaction.

Other related works make use of statically cached programming resources to accelerate data retrieval. For example, the approaches presented in [14, 15] rank output code blocks by normalizing a sigmoid function of the average StackOverflow vote count from a June 2013 static data dump. Selene [20] recommends equivalent code blocks by searching a repository of 2 million example programs to provide usage examples for a given input code block. Sourcerer [21] is another code search engine for a large-scale code repository (SourceForge). Strathcona [22], similarly to the systems above, also uses a repository-based search corpus and provides the user with a structural overview of relevant code rather than actual code examples and discussions. A recommendation system developed by Bacchelli et al. [11] utilizes a vector space model with *tf-idf* as its frequency weighting model along with a singular query corpus source of StackOverflow. Seahawk [16] also uses a static and publicly available dump of StackOverflow questions and lacks support for Swift.

As mentioned in subsection 2.2 above, StackOverflow receives close to 6,000 new questions a day. A static data snapshot from 2 years ago may be sufficient to mine for information about established language ecosystems and environments.

However, the focus of `native-2-native` is mobile computing with rapidly evolving programming environments and language ecosystems. By combining vector space and linear models on live StackOverflow data, `native-2-native` returns suggestions that are more relevant, up-to-date, and better geared toward newer languages, such as Swift. Seahawk motivates the necessity of using the static dump to be able to search by means of the *Apache Solr* search system to achieve high performance efficiency. Although it would be unrealistic to try and match the performance efficiency of searching against a static local snapshot, we discovered that carefully calibrating weights and feature sets for the *tf-idf* analysis and vector space model not only provides complete and relevant results for both StackOverflow posts and other code sources, but also yields performance levels sufficient for practical use. Lastly, [18] focuses on providing useful documentation that supersedes standard API usage documentation. While an important aspect of the developer's ability to create mobile applications can at times include understanding the necessary documentation, `native-2-native` focuses instead on the code synthesis and not just on supplemental documentation for the developer.

The recentness of the Swift language's entry into the mobile computing space renders mainstream native transpilation (i.e., source-to-source compilation) systems inapplicable. For example, Google's J2ObjC [23] converts pure Java source code into Objective-C source code. Although a powerful and practical tool used by Google internally, J2ObjC lacks support for Android APIs and the controller component of the MVC design pattern, as well as converting to Objective-C rather than the new standard of Swift. J2ObjC is designed to convert Java business logic into Objective-C and while this approach is quite useful for cross-platform applications that are data intensive, it lacks in its ability to provide support for native mobile APIs. By contrast, `native-2-native` embraces the native mobile APIs such as Android and iOS. While it remains unclear whether rule-based compiler translation is even capable of bridging the differences between the platforms as architecturally dissimilar as Android and iOS, in the meantime `native-2-native` provides a practical solution for deriving working

Swift code analogous to its Android counterpart.

Cross-platform mobile frameworks, such as ReactNative [24], Cordova [25] and PhoneGap [26], are increasingly gaining traction, as supporting cross-platform applications is rapidly becoming required for any non-trivial mobile venture. Cordova and PhoneGap are designed to produce "web view" based applications for a mobile platform by allowing the developer to write the base application in HTML/Javascript/CSS and then to display the application's UI components using a provided special framework. ReactNative on the other hand, renders this base application code into native platform widgets [27] that call native APIs directly. However, such frameworks are specifically designed to aid in the development of UIs and views. Hence, they lack support for native mobile platform APIs that control other platform resources, such as sensors and networking. By contrast, the `native-2-native` approach provides the flexibility, opportunity, and freedom to use any custom widgets and hardware specific APIs for cross platform applications, without constraining the developer to a set of pre-determined widgets, for which it would be impossible to cover all possible application scenarios of using native APIs.

A representative of yet another approach to support the development of cross-platform mobile applications is Mobl [28], which provides a domain-specific language to express mobile functionality in a platform-independent fashion. Similarly to the cross-platform mobile frameworks discussed above, Mobl also relies on web application based architectures to simulate native application deployment on mobile devices. Hence, it necessarily has to limit access to certain hardware services (e.g., low-level network interfaces). Despite its several promising research ideas, the Mobl approach is not quite ready for practical use, as its prototype implementation leaves out some language details, such as the type checker not supporting all properties. Another impediment to practical adoption is performance, with the non-native end application having to continuously communicate with a server-based cache. By contrast, even our reference implementation of the `native-2-native` approach can provide immediate practical value to the mobile developer by automating the most laborious aspects of searching the web for relevant code snippets.

Nguyen et al. [29] present a statistical semantic language model for source code (SLAMC), an extendable language model that leverages the lexical information and local context of code tokens. By breaking up the code into tokens and patterns, this model can aid in language translation and migration. We could have made use of this model in `native-2-native` to produce search queries by analyzing code input. However, the language design differences between Java and Swift would likely render this model inapplicable as the basis for translating between these two languages. In other words, Swift's declarative design makes it hard to produce valuable semantic patterns, while the plethora of available mobile APIs would render any semantic language model ineffective for the `native-2-native` intended usage scenarios.

Mishne et al. present Prime [30], a tool that provides a searchable and consolidated index for API usage. The tool mines and indexes code snippets of a given API to uncover its inner intricacies, with the goal of assisting the developer with using new APIs. Compared to `native-2-native`, Prime focuses on a different problem domain, in which the user is expected to be able to specify both the query and the intended purpose of using the API. By contrast, `native-2-native` automatically generates a search query against a given set of web-based programming resources rather than a batch of API related code snippets.

`Native-2-native` utilizes common themes and features across various prior recommendation systems, but it applies them to a problem that arises from the realities of modern mobile development—the necessity of supporting popular mobile applications on all major platforms, despite the inherent dissimilarities in the platforms' languages, APIs, and architectures. Potentially, this problem may be addressed by state-of-the-art cross-platform source-to-source translation, with some inroads already in place [31]. Nevertheless, precise source-to-source compilation capable of translating Java Android to and from Swift iOS functionality for native APIs remains a futuristic vision.

Finally, by using sources other than just user-driven StackOverflow posts, `native-2-native` mines a wider feature net without being restricted to API mappings [32]. This property allows our approach to provide potentially more valuable and relevant search results based on the developer's code context. Given that the overriding goal of `native-2-native` is to aid the developer by suggesting and synthesizing equivalent code, any new suggested code, as long as it correctly implements some API, is likely to prove useful to the mobile developer.

7. Future Work and Concluding Remarks

This article has presented a novel approach for automatic code synthesis from Android/Java to iOS/Swift and vice versa by utilizing popular web-based programming resources. To enable our approach, we first gleaned several insights underlying the realities of modern mobile software development and the mobile computing market. Our approach is concretely realized as `native-2-native`, which includes primarily extracting core functionality from input Java or Swift source code, searching, ranking, and code synthesis. The reference implementation of `native-2-native` is an Eclipse plug-in that allows the developer to select input mobile source code and choose from the top two returned search results before using this selection to synthesize the output target mobile source file.

Since the approach is bidirectional, both Android and iOS developers can take advantage of this system to ease their effort in developing a cross platform mobile application. The evaluation showed that 74% of code block results from Java $\rightarrow$ Swift and 91% of code block results from Swift $\rightarrow$ Java produce useful and intended functionality for the subject native API experiments. These results indicate that the presented approach can become a pragmatic and valuable programming tool in the arsenal of mobile software developers.

One potential direction for future work entails reusing the search data in an intelligent manner. Our current approach relies on generating a document corpus upon each query, but these results are discarded for all future queries, not just locally for a particular user but across all users. If instead, these meta-data objects were saved globally to become accessible for all users, then our approach could be further optimized in the following two major ways. First, the potential use of a preference server would facilitate an ability to predict the semantics of future queries made by a particular user based on previous inquiries they made. This predicted potential query could be weighted along with the newly generated meta-data object and form the user's next query. Secondly, we could store all meta-data objects across all users in a set of online clusters. The clusters would be defined by some similarity measure related to the original ranking model. This optimization would facilitate a speedup when a user's query is within the cluster and within some threshold of similarity specified by the user, as one then would not need to continue a full web search for potential equivalent Swift/Java code blocks.

Another potential avenue for improvement is to incorporate the translation of the "Model" component of an application's Model-View-Controller pattern. This enhancement could be accomplished by integrating with `native-2-native` the previously discussed [23] or another source-to-source translator of non-Android Java code. Similarly, we can further improve the translation and ranking by allowing users to rate the code block returned by the algorithm. These ratings would then be incorporated into future queries not just for this user but for all users making similar queries.

Yet another future work direction would expand the number of target platforms to Windows Phone and platform-independent JavaScript frameworks such as PhoneGap [33]. The developer would implement a new feature on one platform and then get suggestions for all the other supported platforms. Comparing the software metrics of the equivalent code blocks on different platforms can shed light on various software engineering properties of different languages and architectures.

Finally, another interesting direction for future work concerns the query component of `native-2-native`. As shown in Section 4, while for the majority of test cases (85% when converting to Swift and 92% when converting to Java) the queries that we create can retrieve at least one relevant answer, we still miss quite a few cases. In the future, we plan to explore the exact causes of the cases in which our query generation fails and develop methods for improving the generation of queries. In particular, we plan to increase the effectiveness of our query generation algorithm by applying Natural Language Processing techniques. These techniques have potential to increase the precision of selecting relevant query terms as well as to enrich the queries by applying paraphrasing and assigning linguistic structures to the resulting queries.

Availability. `native-2-native` is available from `https://github.com/antuanb/Native-2-Native`. The site includes the full source code for the approach, including the integration with Eclipse, open-source license, detailed results, and additional evaluation use cases.

8. Acknowledgments

This research is supported in part by the National Science Foundation through Grant CCF-1116565. We would like to thank Brendan Avent, Young-Woo Kwon, and Myoungkyu Song for their insightful comments and suggestions for improving the manuscript.

References

[1] ISO/IEC: Software engineering–software life cycle processes–maintenance, Tech. rep., International Organization for Standardization, Geneva, Switzerland (2006).

[2] A. Byalik, S. Chadha, E. 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, ACM, New York, NY, USA, 2015, pp. 99–108. `doi:10.1145/2814204.2814210`. URL `http://doi.acm.org/10.1145/2814204.2814210`

[3] StackOverflow.com, Stack overflow (2015).

[4] StackOverflow.com, Usage of /info - stack exchange api (2015).

[5] StackOverflow.com, Stack overflow developer survey 2015 (2015).

[6] F. P. Brooks, No silver bullet: Essence and accidents of software engineering, Computer 20 (4) (1987) 10–19.

[7] www.tiobe.com, Usage of tiobe language use statistics (2015).

[8] P. Jackson, I. Moulinier, Natural language processing for online applications. Text retrieval, extraction and categorization, Vol. 5 of Natural Language Processing, Benjamins, 2002.

[9] L. Breiman, Bagging predictors, Machine Learning 24 (2) (1996) 123–140. doi:10.1007/BF00058655.

[10] G. Salton, A. Wong, C. S. Yang, A vector space model for automatic indexing, Commun. ACM 18 (11) (1975) 613–620. doi:10.1145/361219.361220.

[11] A. Bacchelli, L. Ponzanelli, M. Lanza, Harnessing stack overflow for the IDE, in: Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering, RSSE '12, IEEE Press, 2012, pp. 26–30.

[12] M. Robillard, R. Walker, T. Zimmermann, Recommendation systems for software engineering, Software, IEEE 27 (4) (2010) 80–86. doi:10.1109/MS.2009.161.

[13] M. P. Robillard, W. Maalej, R. J. Walker, T. Zimmermann (Eds.), Recommendation Systems in Software Engineering, Springer, 2014. doi:10.1007/978-3-642-45135-5.

[14] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, M. Lanza, Prompter: A self-confident recommender system, in: Proceedings of the 2014 IEEE

International Conference on Software Maintenance and Evolution, ICSME '14, IEEE Computer Society, 2014, pp. 577–580. `doi:10.1109/ICSME.2014.99`.

[15] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, M. Lanza, Mining stack-overflow to turn the ide into a self-confident programming prompter, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, 2014, pp. 102–111. `doi:10.1145/2597073.2597077`.

[16] L. Ponzanelli, A. Bacchelli, M. Lanza, Seahawk: Stack Overflow in the IDE, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE, 2013, pp. 1295–1298.

[17] A. Zagalsky, O. Barzilay, A. Yehudai, Example overflow: Using social media for code recommendation, in: Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on, 2012, pp. 38–42. `doi:10.1109/RSSE.2012.6233407`.

[18] J. Kim, S. Lee, S.-w. Hwang, S. Kim, Towards an intelligent code search engine., 2010.

[19] B. Vasilescu, V. Filkov, A. Serebrenik, Stackoverflow and github: Associations between software development and crowdsourced knowledge, in: Social Computing (SocialCom), 2013 International Conference on, 2013, pp. 188–195. `doi:10.1109/SocialCom.2013.35`.

[20] W. Takuya, H. Masuhara, A spontaneous code recommendation tool based on associative search, in: Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, SUITE '11, 2011, pp. 17–20. `doi:10.1145/1985429.1985434`.

[21] S. Bajracharya, J. Ossher, C. Lopes, Sourcerer: An internet-scale software repository, in: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, SUITE

'09, IEEE Computer Society, 2009, pp. 1–4. doi:10.1109/SUITE.2009.5070010.

[22] R. Holmes, G. C. Murphy, Using structural context to recommend source code examples, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, 2005, pp. 117–125. doi:10.1145/1062455.1062491.

[23] J2ObjC.org, J2objc (2015).

[24] https://facebook.github.io/react native/, React native — a framework for building native apps using react (2016).

[25] https://cordova.apache.org/, Apache cordova (2016).

[26] http://phonegap.com/, Phonegap (2016).

[27] E. Holmes, T. Bray, Getting Started with React Native, Packt Publishing, 2015.
URL https://books.google.com/books?id=vSLlCwAAQBAJ

[28] Z. Hemel, E. Visser, Declaratively programming the mobile web with mobl, in: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, ACM, Portland, Oregon, USA, 2011.

[29] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, A statistical semantic language model for source code, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 532–542. doi:10.1145/2491411.2491458.
URL http://doi.acm.org/10.1145/2491411.2491458

[30] A. Mishne, S. Shoham, E. Yahav, Typestate-based semantic code search over partial programs, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, ACM, New York, NY, USA, 2012, pp. 997–1016.

doi:10.1145/2384616.2384689.
URL http://doi.acm.org/10.1145/2384616.2384689

[31] A. Puder, O. Antebi, Cross-compiling android applications to ios and windows phone 7, Mob. Netw. Appl. 18 (1) (2013) 3–21. doi:10.1007/s11036-012-0374-2.

[32] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, Q. Wang, Mining api mapping for language migration, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, 2010, pp. 195–204. doi:10.1145/1806799.1806831.

[33] R. Ghatol, Y. Patel, Beginning PhoneGap: Mobile Web Framework for JavaScript and HTML5, 2012.