CMPT 497 Capstone Project Report

Babel Swift: An Objective-C to Swift Conversion Tool

by

Dongyuan Liu

School of Computing Science
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Spring 2016

Abstract

Swift is a new programming language developed by Apple, designed to replace Objective-C as the main programming language on Apple platforms. This report introduces Babel Swift, a new conversion tool for converting Objective-C code to Swift. Our tool focuses on converting code snippets. The most important feature of Babel Swift is the ability to convert incomplete code snippets with undefined identifiers. In the report, we describe the design and implementation of Babel Swift. We also evaluate Babel Swift by comparing with other existing tools.

Keywords: Objective-C, Swift, Source Code Transformation

Table of Contents

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Objective-C and Swift

Objective-C is a programming language which adds Smalltalk's object-oriented features to the C programming language, designed by Brad Cox and Tom Love in 1983 [24]. Objective-C allows programmers to use all the features in the C programming language, together with features of the object-oriented paradigm (classes, objects, instance and class methods, etc.). It is the primary programming language for writing software on Apple's OS X and iOS platforms [9].

In 2014, Apple introduced Swift, a new programming language for building iOS, OS X, watchOS, and tvOS apps. Swift adopts safe programming patterns and modern features, makes writing programs easier and more flexible [18]. Some additional features of Swift include tuples, generics, functional programming features, built-in error handling, and advanced control flow (`do`, `guard`, `defer`, and `repeat` keywords). It also has a new keyword `let` to define constants, and a new feature known as *optionals* to ensure safety. Because of its advanced design, Swift is becoming more and more popular. According to a developer survey by Stack Overflow, a popular question-and-answer website for developers, in 2016, Swift is the third trending technology [17]. However, there are a large number of existing source code in Objective-C. On GitHub, a popular website for hosting source code, by April 2016, there are 36,568 active Objective-C repositories, which is more than 3 times as many as 11,138 active Swift repositories [6].

1.2 Objective-C to Swift Conversion Tools

In a Swift project, programmers may want to use existing Objective-C code. Although Objective-C and Swift files can coexist in a single project [21], it is not possible to use Objective-C directly in a Swift source file. To overcome this limitation, programmers usually convert Objective-C code to Swift manually. However, this manual process is not only time

consuming, but also error-prone. Objective-C to Swift conversion tools provide a safe and convenient solution to this problem.

There are several existing conversion tools: objc2swift [8] is an open-source conversion tool. Swiftify [20] and iSwift [5] are two commercial conversion tools.

The limitation of the existing conversion tools is that they are all designed for converting complete source files or complete projects. They have very limited support for incomplete code snippets with undefined identifiers. However, converting incomplete code snippets is a typical use case of conversion tools. For example, a programmer may find an Objective-C code snippet in a Stack Overflow answer, and want to use it in a Swift project.

1.3 Introducing Babel Swift

This report introduces Babel Swift, a new conversion tool for converting Objective-C code to Swift. Babel Swift is designed for handling incomplete code snippets with undefined identifiers. It can also generate code using Swift's new features, e.g. it has the ability to convert C-style `for` loops to `for-in` statements and `stride` in Swift. Babel Swift is written in Python, based on libclang [3]. The full source code can be found on GitHub [23].

1.4 Organization of this Report

The rest of this report is organized as follows. We describe the objectives and implementation in Chapter 2. We briefly discuss the alternative approaches in Chapter 3. In Chapter 4, we evaluate our conversion tool, and compare it with other existing tools. In Chapter 5, we conclude the report.

Chapter 2

Objectives and Implementation

Babel Swift is an Objective-C to Swift code conversion tool. The main objective is to convert incomplete code snippets.

2.1 Overall Design

Babel Swift is written in Python, based on libclang [3] and its Python binding. libclang is a C interface to Clang [2], a C language family frontend for LLVM. libclang provides a small API for parsing source code using Clang and accessing the parsed abstract syntax tree (AST). By using Clang as the parser, we can focus on improving the conversion itself.

In libclang, a *cursor* represents a location within the AST. Our transform function takes the root cursor as input, and outputs the converted Swift code. We use a depth-first traversal to recursively visit all the nodes in the AST. We will discuss how we generate the AST, and how we generate the Swift code from the AST in the following sections.

2.2 Handling Code Snippets

In this report, we use the term *code snippet* to represent a piece of code *within* a function. A code snippet consists of one or more statements. A code snippet can not contain any function definitions. Listing 1 is an example of Objective-C code snippet (`NSPopover` is a class in *AppKit*, a foundational framework used in Mac development).

```
NSInteger answer = 42;
if (answer == 42) {
    NSPopover *popover = [[NSPopover alloc] init];
    [popover run];
}
```

Listing 1: A code snippet

Although it seems to be easy to handle code snippets, libclang refuses to parse them. For Objective-C, libclang expects the input to be a complete source file. To overcome this limitation, Babel Swift wraps the input code snippet into a complete source file. In the example above, the wrapped input for libclang is:

```
#import <AppKit/AppKit.h>
#import "BabelSwiftHeader.h"

@implementation __BABEL_SWIFT_WRAPPER_CLASS__

- (id)__BABEL_SWIFT_WRAPPER_METHOD__ {

NSInteger answer = 42;
if (answer == 42) {
    NSPopover *popover = [[NSPopover alloc] init];
    [popover run];
}

}

@end
```

Listing 2: A complete source file containing the snippet in Listing 1

We use two special names `__BABEL_SWIFT_WRAPPER_CLASS__` and `__BABEL_SWIFT_WRAPPER_METHOD__` to wrap the snippet. `@implementation __BABEL_SWIFT_WRAPPER_CLASS__` and `@end` define a class, and `- (id)__BABEL_SWIFT_WRAPPER_METHOD__` defines a method. We also import two headers: `AppKit/AppKit.h` contains the fundamental declarations in Cocoa[1], e.g. `NSInteger` and `NSDictionary`. `BabelSwiftHeader.h` is for handling *incomplete code snippets*, which will be elaborated in the next section.

2.3 Handling Incomplete Code Snippets

We define an *incomplete code snippet* as a code snippet which contains undefined identifiers. For example, the following Objective-C code snippet is an *incomplete code snippet*, because both `speed` and `TimeMachine` are undefined identifiers.

[1] Apple's native object-oriented API for its OS X operating system

```
if (speed == 88) {
    TimeMachine *timeMachine = [[TimeMachine alloc] init];
    [timeMachine run];
}
```

Listing 3: An incomplete code snippet

In Clang, the Sema module is responsible for both semantic analysis and AST construction. Thus, libclang cannot generate the AST for incomplete code snippets because it does not know whether an identifier is a variable name or a class name. For the above snippet, libclang generated the following error messages:

```
input.m:7:5: error: use of undeclared identifier 'speed'
input.m:8:5: error: use of undeclared identifier 'TimeMachine'
input.m:8:18: error: use of undeclared identifier 'timeMachine'
input.m:8:34: error: use of undeclared identifier 'TimeMachine'
input.m:9:6: error: use of undeclared identifier 'timeMachine'
```

Listing 4: Error messages generated by libclang for snippet in Listing 3

As mentioned before, we included a special header file `BabelSwiftHeader.h`. We can fix the errors by adding pseudo-declarations in this header file. For each undefined variable name, we declare an empty class interface, and declare the variable using that new class. For each undefined class name, we add an empty class interface in the header file. After fixing all the errors, libclang would be able to generate the AST. The header file `BabelSwiftHeader.h` for Listing 3 is like:

```
@class __BABEL_SWIFT_WRAPPER_CLASS__;
@class __BABEL_SWIFT_PSEUDO_CLASS_1__;
@class TimeMachine;

@interface __BABEL_SWIFT_WRAPPER_CLASS__ : NSObject
@end

@interface __BABEL_SWIFT_PSEUDO_CLASS_1__ : NSObject
@end

@interface TimeMachine : NSObject
@end

__BABEL_SWIFT_PSEUDO_CLASS_1__ *speed;
```

Listing 5: The content of `BabelSwiftHeader.h` after fixing all the errors

In the above listing, the three `@class`es are forward declarations. The three pairs of `@interface` and `@end` are the empty class interfaces.

We use multiple iterations to fix all the errors. For each iteration, we only try to fix the first error reported by libclang. We use the identifier name to guess whether it is a class name or a variable name. If the identifier starts with a capital letter, we assume it is a class name, otherwise we assume it is a variable name. It is obvious to see that we may make wrong guesses, e.g. a programmer could name a variable `IPAddress`. To make sure all the identifiers are being declared correctly, if it failed to fix the error, we will make another opposite guess. We repeat this approach until there are no errors. The identifiers could have other uses than class names or variable names (e.g. macros and C function names), we currently cannot convert snippets with such identifiers.

2.4 Generating Swift Code from AST

After wrapping the snippet into a complete file and fixing all the errors, we use libclang to parse the file and generate the AST. In libclang, each node in the AST is called a *cursor*. Our transform function takes a cursor as input, and outputs the converted Swift code. We use a depth-first traversal to recursively visit all the nodes in the AST.

The AST generated by libclang for Listing 3 would be:

```
(CursorKind.COMPOUND_STMT)
 +-- (CursorKind.IF_STMT)
    +--== (CursorKind.BINARY_OPERATOR)
    |  +-- (CursorKind.IMPLICIT_CAST_EXPR_STMT)
    |  |  +--speed (CursorKind.DECL_REF_EXPR)
    |  +-- (CursorKind.IMPLICIT_CAST_EXPR_STMT)
    |     +-- (CursorKind.INTEGER_LITERAL)
    +-- (CursorKind.COMPOUND_STMT)
       +-- (CursorKind.DECL_STMT)
       |  +--timeMachine (CursorKind.VAR_DECL)
       |     +--TimeMachine (CursorKind.OBJC_CLASS_REF)
       |     +--init (CursorKind.OBJC_MESSAGE_EXPR)
       |        +--alloc (CursorKind.OBJC_MESSAGE_EXPR)
       |           +--TimeMachine (CursorKind.OBJC_CLASS_REF)
       +--run (CursorKind.OBJC_MESSAGE_EXPR)
          +-- (CursorKind.IMPLICIT_CAST_EXPR_STMT)
             +--timeMachine (CursorKind.DECL_REF_EXPR)
```

Listing 6: The AST generated by libclang for Listing 3

We use the notation as follows to describe how the visitor generates code for different cursor types: The left side is the type of the cursor, the right side the the code returned by the visitor. On the right side, `monospaced` characters are literal in the output. *italic* text means that it needs to be executed to get the value. C_i represents the i-th child of the cursor. *visit(c)* represents the code generated by visiting cursor *c*. *this* is the current cursor we are visiting, it has some properties provided by libclang. *this.spelling* is the name of the current cursor. *this.cstyleCastTargetType* is the name of the target type in a C-style cast.

Table 2.1: Translation scheme of the visitor

Cursor Type	**Generated Code**
`DeclRefExpr`, `ObjCClassRef`	*this.spelling*
`ImplicitCastExprStmt`	*visit*(C_1)
`ObjCBoxedExprStmt`	*visit*(C_1)
`IntegerLiteral`, `FloatingLiteral`	*this.spelling*
`ObjCSelfExpr`	*this.spelling*
`ObjCStringLiteral`	*this.spelling*
`ObjCArrayLiteralStmt`	`[` *visit*(C_1) `,` *visit*(C_2) `,` ... `,` *visit*(C_n) `]`
`ObjCDictionaryLiteralStmt`	`[` *visit*(C_1) `:` *visit*(C_2) `,` ... `,` *visit*(C_{n-1}) `:` *visit*(C_n) `]`
`CStyleCastExpr`	*visit*(C_0) `as` *this.cstyleCastTargetType*
`DeclStmt`	`let` *this.spelling* `=` *visit*(C_1)
`ParenExpr`	`(` *visit*(C_1) `)`
`IfStmt`	`if` *visit*(C_1) *visit*(C_2)
`WhileStmt`	`while` *visit*(C_1) *visit*(C_2)
`BinaryOperator`	*visit*(C_1) *this.spelling* *visit*(C_2)
`UnaryOperator`	*this.spelling* *visit*(C_1)
`MemberRefExpr`	*visit*(C_2) `.` *visit*(C_1)
`CompoundStmt`	`{` *visit*(C_1) *visit*(C_2) ... *visit*(C_n) `}`
`CallExpr`	*visit*(C_1) `(` *visit*(C_2) `,` ... `,` *visit*(C_n) `)`

After applying the translation scheme, we get the following Swift code from the above AST:

```
if speed == 88 {
   let timeMachine = TimeMachine()
   timeMachine.run()
}
```

Listing 7: Swift code generated from AST in Listing 6

2.5 Special Cases

The translation scheme for most of the cursor types is not complex and can be described by the notation we used. There are a few special cases that need different strategies.

2.5.1 Swift Data Types

Swift has a different set of data types than Objective-C. We use mappings to convert primitive data types. Because it is allowed to use C types in Objective-C, we also have mappings for C types. Once we see an Objective-C or C data type in the mapping, we use the Swift counterpart in the output.

Table 2.2: Mappings for primitive data types

Objective-C [1]	**Swift [19]**	**Size in 32/64-bit runtime (bytes)**
`char`	`Int8`	1 / 1
`BOOL`, `bool`	`Int8`	1 / 1
`short`	`Int16`	2 / 2
`int`	`Int32`	4 / 4
`long`	`Int`	4 / 8
`long long`	`Int64`	8 / 8
`size_t`	`Int`	4 / 8
`time_t`	`Int`	4 / 8
`NSInteger`	`Int`	4 / 8
`NSUInteger`	`Int`	4 / 8
`float`	`Float`	4 / 4
`double`	`Double`	8 / 8

Enumeration types also need special conversion. Due to its complexity, we do not support enumerations yet.

2.5.2 Method Calls

In Objective-C, the syntax for method calls is different than other C-like languages. For example, the following code would call `- insertSubview:atIndex:` with parameters `mySubview` and `2`. The syntax is similar to *named arguments* in other languages, except that the order of the arguments cannot be changed.

```
[view insertSubview:mySubview atIndex:2];
```

Swift uses dot syntax for method calls. The counterpart in Swift is much more like in other languages [22]:

```
view.insertSubview(mySubview, atIndex: 2)
```

Babel Swift recognizes method calls in Objective-C, extracts the method signatures and arguments, and converts them to Swift method calls.

2.5.3 Initializations

In Objective-C, class initializers must begin with `init`, or `initWith` if they take arguments [4]. Programmers also need to call `alloc` to allocate memory explicitly when initializing an object. A complete object initialization in Objective-C is like:

```
UITableView *tableView = [[UITableView alloc] initWithFrame:CGRectZero
    style:UITableViewStyleGrouped];
```

In Swift, the `init` or `initWith:` is omitted. `alloc` does not need to be called explicitly. The object initialization becomes:

```
let tableView: UITableView = UITableView(frame: CGRectZero, style: .Grouped)
```

2.5.4 `for` Loops

`for` loops are widely used in Objective-C. A typical `for` loop in Objective-C is like:

```
for (int index = 0; index < array.count; index++) {
    NSLog(@"%@", array[index]);
}
```

Listing 8: A typical Objective-C `for` loop

However, C-style `for` loops are deprecated and will be removed completely in Swift 3.0 [11]. To follow the best practice in Swift, Babel Swift would generate the following Swift code for the above `for` loop:

```
for index in 0..<array.count {
    NSLog("%@", array[index])
}
```

Listing 9: Swift code converted from Listing 8

An even better way in terms of coding style for this case is to eliminate `index` completely, as shown in Listing 10. However, it requires Babel Swift to look into the `for` body. This introduces much more complexity and we decided not to do this transformation.

```
for elem in array {
    NSLog("%@", elem)
}
```

Listing 10: An better Swift `for` loop for Listing 8

The C-style `for` loop requires three parts: Initialization, condition, and afterthought. Currently we only support converting `for` loops with the following constraints: the initialization part must be a single variable definition; the condition part must be a boolean operation between the variable defined in the initialization part and an expression, the operator must be `<`, `<=`, `>`, or `>=`; the afterthought part must be an increment or decrement of the variable defined in the initialization part.

2.6 Remaining Issues

As mentioned before, Babel Swift cannot handle undefined identifiers other than class names or variable names. Besides, Babel Swift can only convert `for` loops with certain constraints.

Comments are useful in source code. It would be good to keep the comments during the conversion. Currently, they are simply dropped due to the limitation of libclang. This could be implemented in the future by adding the functionality to keep the comments in libclang.

Another potential improvement is to support larger forms of code like functions and classes in addition to code snippets. It could use the same approach to allow undefined identifiers in functions and classes.

Chapter 3

Alternative Approaches

In this chapter, we briefly discuss some alternative approaches to implement the conversion tool. Instead of using libclang, we could also build the tool directly upon Clang (or with LibTooling), build the tool based on objc2swift, or write the parser from scratch.

Babel Swift is based on libclang. As noted before, libclang offers a convenient and stable API for accessing the AST generated by Clang. It also has an official Python binding, which is very helpful for fast prototyping. However, libclang does not provide full access over the AST. To implement a decent conversion tool, many new features had to be added in libclang. Moreover, as a young addition to Clang, libclang does not have complete documentation.

Because Apple is the main user and driver behind Objective-C and Apple uses Clang, Clang is the de facto standard frontend for the language. Using a Clang-based approach (including libclang and LibTooling) could save a lot of effort on parsing the Objective-C code.

All the approaches differ in the frontend. Once the code in source language is converted to AST, the implementations of the backend are almost the same.

3.1 Building Directly Upon Clang

Building the tool directly upon Clang provides full control over the AST. There are two major drawbacks: First, Clang itself is not designed to be used as a library. Setting up Clang to use it as a library is extremely complex. Second, Clang is evolving rapidly. It does not promise a stable API. If the tool is building upon a certain version of Clang, there is a good chance that the code would not work with the next version. In addition, the only possible language to implement based on Clang is C++.

3.1.1 LibTooling

LibTooling is a library for writing standalone tools based on Clang [7]. As noted before, Clang has a complex setup process. Instead, LibTooling offers a straightforward setup process. LibTooling is an ideal framework to develop tools based on Clang.

Despite the drawbacks (unstable API, cannot use languages other than C++), if the objective is to build a conversion tool which covers the complete Objective-C syntax, LibTooling is a better choice than libclang because of its full control over the AST.

3.2 Building Based on objc2swift

objc2swift [8] is an open-source Objective-C to Swift conversion tool. A viable way to implement Babel Swift is to make improvements based on objc2swift. However, objc2swift was released after we started the project.

3.3 Custom Parser

Using a custom parser provides the best flexibility over the implementation. The parser could be written by hand or generated by parser generators. The most obvious drawback is that it needs a large amount of effort to implement a complete parser. objc2swift [8] is using this approach.

Chapter 4

Evaluation

Evaluating a code conversion tool is difficult, because there are no obvious metrics for them. Typical metrics like performance are not our focus. Instead, the three qualities for code conversion tools we are focusing on are *support for input source code*, *correctness of conversion*, and *quality of converted code*. This chapter defines these terms and attempts to evaluate these qualities in the following sections.

4.1 Methodology

To collect real world Objective-C code for evaluation, we chose most voted questions under **objective-c** tag on Stack Overflow. If the question and its answers contain code, we can use them for our evaluation. We have collected 6 pieces of code in total: *dispatch.m*, *location.m*, *prepareforsegue.m*, *sortusingblock.m*, *textfield.m*, and *viewwillappear.m*. See Appendix A for all the code we have collected.

Because Babel Swift only supports converting code snippets, we did the two steps to preprocess the code:

1. If the code contains more than one function, we only keep the first function and exclude anything outside that function.

2. If the code contains a function, we use the function body as the input for Babel Swift; we use the whole function as the input for other conversion tools.

After preprocessing, we run Babel Swift, objc2swift, Swiftify, and iSwift on all the 6 inputs. Since Swiftify and iSwift are commercial tools, we use their free online versions for evaluation.

4.2 Support of Input Source Code

If a conversion tool succeeded in converting a piece of source code, we say that it *supports* that piece of code. Table 4.1 shows whether a conversion tool support a certain piece of code.

Table 4.1: Support of input source code

Code Piece	Babel Swift	objc2swift	Swiftify	iSwift
dispatch	Yes	No	Yes	Yes
location	Yes	No	Yes	No
prepareforsegue	Yes	No	Yes	No
sortusingblock	No	No	Yes	No
textfield	No	No	Yes	No
viewwillappear	Yes	No	Yes	No
Total	4 (66.7%)	0 (0%)	6 (100%)	1 (16.7%)

Babel Swift fails to convert *sortusingblock* because it does not support the block syntax [10] in Objective-C. It fails to convert *textfield* because it does not support properties for primitive types.

4.3 Correctness of Conversions

If the converted code is a valid Swift code, and it has the same functionality as the original code, we say that the conversion is *correct*. It is hard to write a general purpose tool to test the code automatically. So we evaluated the correctness of conversions by hand. Table 4.2 shows the correctness of conversions for each code piece.

Table 4.2: Correctness of conversions

Code Piece	Babel Swift	objc2swift	Swiftify	iSwift
dispatch	No	n/a	No	Yes
location	Yes	n/a	Yes	n/a
prepareforsegue	Yes	n/a	Yes	n/a
sortusingblock	n/a	n/a	Yes	n/a
textfield	n/a	n/a	Yes	n/a
viewwillappear	Yes	n/a	Yes	n/a
Total	3 (50%)	0 (0%)	5 (83.3%)	1 (16.7%)

For *dispatch*, the code generated by Babel Swift was not correct because in Swift, casting to numeric types is done by initializing a numeric object using the from value. Babel Swift currently converts all C-style casts using `as` operator.

4.4 Quality of Converted Code

Finally, we convert all the test cases by hand. See Appendix A for all the hand-converted code. We define two pieces of code are *identical* if they have the same parse tree. If the machine-converted code is identical to the hand-converted one, we say that it is a high quality conversion. Table 4.3 shows the quality of converted code for each conversion tool.

Table 4.3: Quality of converted code

Code Piece	Babel Swift	objc2swift	Swiftify	iSwift
dispatch	Not Identical	n/a	Not Identical	Not Identical
location	Identical	n/a	Not Identical	n/a
prepareforsegue	Identical	n/a	Not Identical	n/a
sortusingblock	n/a	n/a	Identical	n/a
textfield	n/a	n/a	Identical	n/a
viewwillappear	Identical	n/a	Not Identical	n/a
Total	3 (50%)	0 (0%)	2 (33.3%)	0 (0%)

4.5 Evaluation Summary

Because the evaluation was not automated, we can not use a big amount of test data. Based on the 6 inputs, we cannot get any statistically significant results. Despite that, Babel Swift's ability is promising. Babel Swift was able to convert 4 code snippets out of 6. In the Swift code generated by Babel Swift, 3/4 are considered identical to hand-converted. Although there is plenty of room for improvements, it does show that Babel Swift can properly convert some real-world snippets.

Chapter 5

Conclusion

This report presents the design and implementation of a new Objective-C to Swift conversion tool. The key features of our tool include the ability to convert incomplete code snippets, as well as generating code that adopt Swift's new features. In the report, we describe the techniques we used to achieve the objectives. We also briefly discuss the alternative approaches. In the future, this tool could be improved to support comments. Another improvement is to support converting larger forms of code including functions and classes. The implementation of Babel Swift is complete and useable. The source code is fully accessible on GitHub [23]. At last, we evaluate the conversion tool. Results show that it can convert 4/6 of the code snippets we have tested, 3 of them were considered optimal.

Bibliography

[1] 64-bit transition guide for cocoa touch: Major 64-bit changes. https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaTouch64BitGuide/Major64-BitChanges/Major64-BitChanges.html.

[2] clang: a c language family frontend for llvm. http://clang.llvm.org.

[3] clang: libclang: C interface to clang. http://clang.llvm.org/doxygen/group__CINDEX.html.

[4] Concepts in objective-c programming: Object initialization. https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Initialization/Initialization.html.

[5] Convert objective-c to swift online | iswift. http://iswift.org.

[6] Githut - programming languages and github. http://githut.info.

[7] Libtooling - clang 3.9 documentation. http://clang.llvm.org/docs/LibTooling.html.

[8] objc2swift - the open source obj-c to swift converter. http://objc2swift.yahoo-labs.jp.

[9] Programming with objective-c: About objective-c. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.

[10] Programming with objective-c: Working with blocks. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html.

[11] Remove c-style for-loops with conditions and incrementers. https://github.com/apple/swift-evolution/blob/master/proposals/0007-remove-c-style-for-loops.md.

[12] Stack overflow #1126726. http://stackoverflow.com/questions/1126726/how-to-make-a-uitextfield-move-up-when-keyboard-is-present.

[13] Stack overflow #24062509. http://stackoverflow.com/questions/24062509/location-services-not-working-in-ios-8.

[14] Stack overflow #4139219. http://stackoverflow.com/questions/4139219/how-do-you-trigger-a-block-after-a-delay-like-performselectorwithobjectafter.

[15] Stack overflow #5210535. http://stackoverflow.com/questions/5210535/passing-data-between-view-controllers.

[16] Stack overflow #805547. http://stackoverflow.com/questions/805547/how-to-sort-an-nsmutablearray-with-custom-objects-in-it.

[17] Stack overflow developer survey 2016 results. http://stackoverflow.com/research/developer-survey-2016#technology-trending-tech-on-stack-overflow.

[18] The swift programming language: About swift. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/.

[19] The swift programming language: The basics. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html.

[20] Swiftify | objective-c to swift converter. https://objectivec2swift.com/.

[21] Using swift with cocoa and objective-c (swift 2.2). https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html.

[22] Using swift with cocoa and objective-c (swift 2.2): Interacting with objective-c apis. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/InteractingWithObjective-CAPIs.html#//apple_ref/doc/uid/TP40014216-CH4-ID41.

[23] xhacker/babel-swift: Babel swift is an objective-c to swift converter. https://github.com/xhacker/babel-swift.

[24] Brad L Cox. The object oriented pre-compiler: programming smalltalk 80 methods in c language. *ACM Sigplan Notices*, 18(1):15–22, 1983.

Appendix A

Code Used in Evaluation

```
CGFloat time1 = 3.49;
CGFloat time2 = 8.13;

// Delay 2 seconds
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2.0 * NSEC_PER_SEC)),
               dispatch_get_main_queue(), ^{
    CGFloat newTime = time1 + time2;
    NSLog(@"New time: %f", newTime);
});
```

Listing 11: dispatch.m, from [14]

```
let time1 = 3.49
let time2 = 8.13

// Delay 2 seconds
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, int64_t(2.0 * NSEC_PER_SEC)),
               dispatch_get_main_queue()) {
    let newTime = time1 + time2
    NSLog("New time: %f", newTime)
}
```

Listing 12: Hand-converted Swift code for Listing 11

```objc
- (void)startLocationManager
{
    locationManager = [[CLLocationManager alloc] init];
    locationManager.delegate = self;
    locationManager.distanceFilter = kCLDistanceFilterNone; //whenever we move
    locationManager.desiredAccuracy = kCLLocationAccuracyBest;

    [locationManager startUpdatingLocation];
    [locationManager requestWhenInUseAuthorization]; // Add This Line
}
```

Listing 13: location.m, from [13]

```swift
func startLocationManager() {
    locationManager = CLLocationManager()
    locationManager.delegate = self
    locationManager.distanceFilter = .None //whenever we move

    locationManager.desiredAccuracy = .Best
    locationManager.startUpdatingLocation()
    locationManager.requestWhenInUseAuthorization() // Add This Line
}
```

Listing 14: Hand-converted Swift code for Listing 13

```objc
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([segue.identifier isEqualToString:@"showDetailSegue"]) {
        UINavigationController *navController = (UINavigationController *)
            segue.destinationViewController;
        ViewControllerB *controller = (ViewControllerB *)navController.topViewController;
        controller.isSomethingEnabled = YES;
    }
}
```

Listing 15: prepareforsegue.m, from [15]

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject) {
    if (segue.identifier == "showDetailSegue") {
        let navController = segue.destinationViewController as! UINavigationController
        let controller = navController.topViewController as! ViewControllerB
        controller.isSomethingEnabled = true
    }
}
```

Listing 16: Hand-converted Swift code for Listing 15

```
NSArray *sortedArray;
sortedArray = [drinkDetails sortedArrayUsingComparator:^NSComparisonResult(id a, id b) {
    NSDate *first = [(Person *)a birthDate];
    NSDate *second = [(Person *)b birthDate];
    return [first compare:second];
}];
```

Listing 17: sortusingblock.m, from [16]

```
var sortedArray: [AnyObject]
sortedArray = drinkDetails.sortedArrayUsingComparator() { a: AnyObject, b: AnyObject in
    let first = (a as! Person).birthDate()
    let second = (b as! Person).birthDate()
    return first.compare(second)
}
```

Listing 18: Hand-converted Swift code for Listing 17

```
- (void)textFieldDidBeginEditing:(UITextField *)textField {
    // Keyboard becomes visible
    scrollView.frame = CGRectMake(
        scrollView.frame.origin.x,
        scrollView.frame.origin.y,
        scrollView.frame.size.width,
        scrollView.frame.size.height - 215 + 50); // Resize
}
```

Listing 19: textfield.m, from [12]

```
override func textFieldDidBeginEditing(textField: UITextField) {
    // Keyboard becomes visible
    scrollView.frame = CGRectMake(
        scrollView.frame.origin.x,
        scrollView.frame.origin.y,
        scrollView.frame.size.width,
        scrollView.frame.size.height - 215 + 50) // Resize
}
```

Listing 20: Hand-converted Swift code for Listing 19

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    // register for keyboard notifications
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(keyboardWillShow)
                                                 name:UIKeyboardWillShowNotification
                                               object:nil];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(keyboardWillHide)
                                                 name:UIKeyboardWillHideNotification
                                               object:nil];
}
```

Listing 21: viewwillappear.m, from [12]

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    // register for keyboard notifications
    NSNotificationCenter.defaultCenter().addObserver(self,
        selector: #selector(keyboardWillShow),
        name: UIKeyboardWillShowNotification,
        object: nil)
    NSNotificationCenter.defaultCenter().addObserver(self,
        selector: #selector(keyboardWillHide),
        name: UIKeyboardWillHideNotification,
        object: nil)
}
```

Listing 22: Hand-converted Swift code for Listing 21