國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

Objective-C 對 C#之有效轉換

An Objective-C to C# Translator

高峻皓

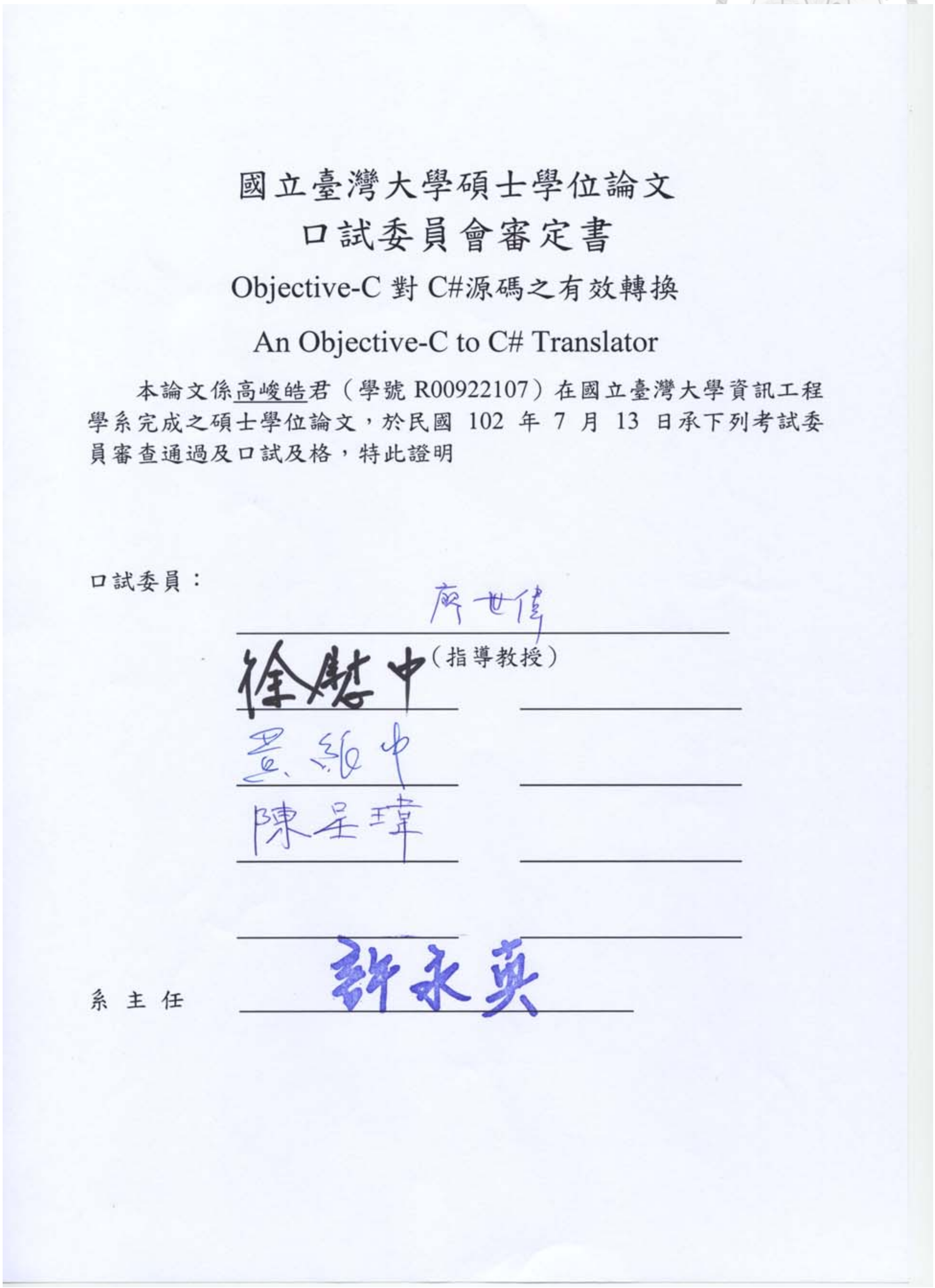
Chun-Hao Kao

指導教授：廖世偉 博士

Advisor: Shih-Wei Liao, Ph.D.

中華民國 102 年 7 月

July, 2013



國立臺灣大學碩士學位論文

口試委員會審定書

Objective-C 對 C#源碼之有效轉換

An Objective-C to C# Translator

本論文係高峻皓君（學號 R00922107）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 102 年 7 月 13 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

廖世偉

徐慰中（指導教授）

[illegible]

陳呈瑋

系 主 任 許永真

誌謝

能完成這篇論文，我要特別感謝我的指導教授廖世偉老師，也要感謝實驗室的每一位成員對我的支持與鼓勵以及新竹工業研究院的學長們給予我的指導和建議，謝謝大家。

中文摘要

隨著手持裝置的流行與發展，數個作業系統成為手持裝置作業系統的主流，對跨平台的支援成為目前佈局於手持裝置的關鍵。在有選擇的情況下，程式開發者不會想要維護兩份程式碼，在本篇論文中的 Objective-C 對 C#轉譯器提供給程式開發者這種選擇: 降低成本，提升系統的維護性，更快速的釋出產品，開發人力的精簡能夠更快速的應變與更精緻的開發小組。本篇論文是第一篇探討 iOS 系統上開發所用到的語言- Objective-C 與一被多種平台所支援之高階語言- C#語法上之差異，並實作一個能將 Objective-C 程式轉換成 C#程式的工具，幫助程式開發者在跨平台開發的時間成本。

Xamarin 不僅限於 Windows Mobile 並在多種平台上支援 C#，但程式開發者仍須維護 Objective-C 的程式碼，此外多數的程式開發者已熟悉 Objective-C，新的語言和在不熟悉的環境上開發學習成本較高，我們的轉譯器減輕了這樣的問題。

ABSTRACT

As the mobile devices are becoming prevalent, two or three operating systems on mobile devices are most dominant. The issue of cross-platform support is critical for comprehensive deployment on mobile devices. No developers want to maintain two codebases if they have a choice. This thesis presents our Objective-C to C# translator that offers developers such a choice: Unifying the codebases or eliminating the need of redundant codebases means cost-reduction, system maintainability, faster product release cycle, and a smaller development team with high agility and affordable expertise requirement, This thesis first investigates the syntactic differences between the Objective-C, one of the programming languages on the iOS system, and another high level language, C#, which is supported on more platforms. Next we present the design and implementation of the translator from Objective-C to C#. The translator helps the developers deploy solutions to a comprehensive set of mobile devices faster and cheaper.

Note that Xamarin now supports C# on multiple platforms beyond just Windows Mobile, but developers still need to deal with existing Objective-C codebases. Furthermore, Objective-C may be more familiar with most mobile developers. Learning curve of a new language and an unfamiliar environment is uncertain and expensive. Our translator alleviates this learning curve issue as well.

CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

With the increasing adoption of mobile devices, more and more people think of the mobile as a part of their life. Because of this, the market of application for mobile developing rapidly; hence great amount of software developers are interested in and devote into this field.

Apple Inc. released the first generation iPhone in 2007 which begins new era of the market of mobile application. Even now, Apple is still leading the market of mobile application. In WWDC 2013, Apple announced that people have downloaded 50 billion apps from the App Store. There are currently 900,000 apps in the store, and 93% of them, are downloaded every month.

As for the developers, the thing most concerned by them is about how many users will make use of the applications which developed by them. Since Apple is the market leader of the mobile devices, iOS platform becomes the first choice of the developers to launch the new product.

As the markets of mobile devices have been noticed, Android and Windows Phone also rise to share this promising market. The openness of Android enable the developers to realize more ideal feature without restraints, while the Windows phone taking advantage from its domination in the PC market which allowed the transfer between desktop and mobile without barriers for user and more easily to synchronize and integrate. As a result, the market of mobile devices was shared by the big three. Hence, for software developers, to ensure that more users would download their applications, developing cross-platform application has become an important issue.

We strive for solving the problem of cross-platform development which encountered by developers and enable them to developed on certain platform but fit on every of them to some extent. Since the number of application on App Store, or the iOS system, is the most among the mobile system and also highly valued by developers which have attracted many of them devoted into, we focus on the resources and developers of iOS system. To make the iOS developer developing on other platform easier, we design a translator and chose Objective-C as source language which is a language especially for development on iOS.

Xamarin provides a solution for cross-platform developing. It offers a single language - C#, a high level and popular language, to develop on three of the most popular mobile platform - iOS, Android, Windows Phone. Because of numbers of developer on Xamarin and maturity, we chose C# as target language which is supported by Xamarin platform.

To lower the barriers to developing on cross-platform, we design a source to source translator for Objective-C to C#. Our tool can be time-saving for developing cross-platform applications. It reduces the routine work between the translations of Objective-C to C#. Speeds up the translation between the Objective-C source code to C#, which enable developers quickly transfer the application on iOS system to other platform leverage Xamarin to achieve the goal for cross-platform development.

The research mainly discuss about the syntax difference between Objective-C and C#, and the translation with the same or similar semantic meaning. Due to preserving the semantics of high-level language, we decided to make the process at high-level layer, build a source to source translator. In this way, the developer could modify the code which is translated easily, because the design of objects consists with the original one.

The rest of the paper is organized as follows. Chapter 2 presents some background knowledge of the translator. Chapter 3 considers issues related to translation from Objective-C to C#. The more details of implementation will at chapter 4. Chapter 5 presents practical translation of two programs in Objective-C. Chapter 6 provides a conclusion and suggestions for future research.

Chapter 2 Background

2.1 Xamarin

Xamarin offers a single language – C#, class library, and runtime that works across all three mobile platforms of iOS, Android, and Windows Phone, while still compiling native applications that has good performance even for games. It contains bindings for nearly the entire underlying platform SDKs in both iOS and Android.

Xamarin offers sophisticated cross-platform support for the three major mobile platforms of iOS, Android, and Windows Phone. Xamarin.Mobile library offers a unified API to access common resources across all three platforms. This can significantly reduce both development costs and time to market for mobile developers that target the three most popular mobile platforms.

Xamarin offers two commercial products, Xamarin.iOS and Xamarin.Android. They're both built on top of Mono. On iOS, Xamarin's Ahead-of-Time Compiler compiles Xamarin.iOS applications directly to native ARM assembly code. On Android, Xamarin's compiler compiles down to Intermediate Language, which is then Just-in-Time compiled to native assembly when the application launches. In both cases, Xamarin applications utilize a runtime that automatically handles things such as memory allocation, garbage collection, underlying platform interop, etc.

2.2 LLVM & Clang

2.2.1 LLVM

The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. LLVM consists of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source project as well as being widely used in academic research.

2.2.2 Clang

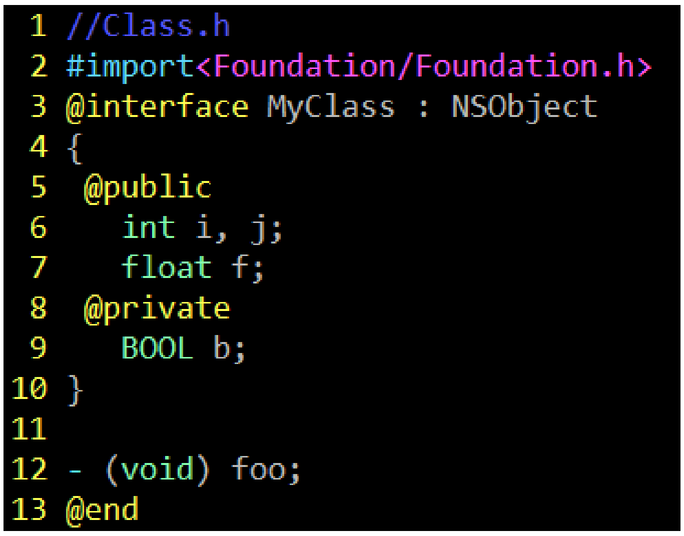
Clang is a subproject of LLVM, and a front-end compiler for the C language family which includes C, C++, Objective-C, and Objective-C++. Clang builds on the LLVM optimizer and code generator, and it provides the necessary components for compiling source code to LLVM IR.

Chapter 3 Problem Overview

With MVC framework in development on iOS, the parts of view and control are binding with GUI in developing, so our tool only translate the syntax for the model part of an Objective-C program into C#.

3.1 Class Interface

In Objective-C, class interface has the same meaning as class definition. The syntax of class interface is in Figure 3.1



```
//Class.h
#import<Foundation/Foundation.h>
@interface MyClass : NSObject
{
 @public
   int i, j;
   float f;
 @private
   BOOL b;
}

- (void) foo;
@end
```

Figure 3.1 Class interface in Objective-C

Class interface consists of two parts, data fields and member functions.

3.1.1 Data Field

In the part of data field we should handle the access control of the field, return type, and the variable name. The translated variable name and access control should be consistent with the original program which is in Objective-C.

3.1.2 Member Function

In the member function, there are four major parts to handle.

a. Instance method or class method

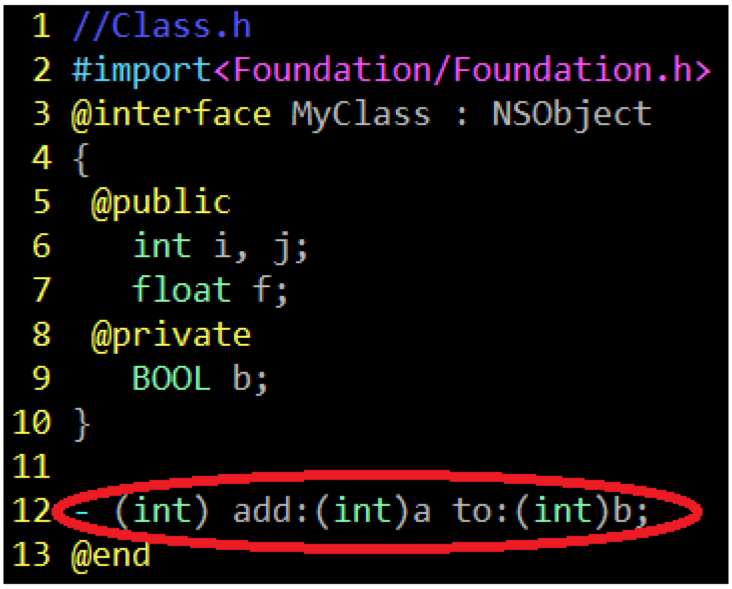
In Objective-C, the first character of function is always “-” or “+”. “-” means this function is a instance method, and “+” means that the function is a class method. The translation should handle this.

b. Access control

Access control is determined by if the method is declared in interface section or not. If the method is declared in the interface section, the access control of this function should be public. Otherwise, the method is just defined in implementation section, and then the function should be private. There is no “protected” in Objective-C which is available in java and C++.

c. Function Name

The name of functions also called “selector” in Objective-C. Selector in Objective-C has a particular format with the number of arguments which is more than one.

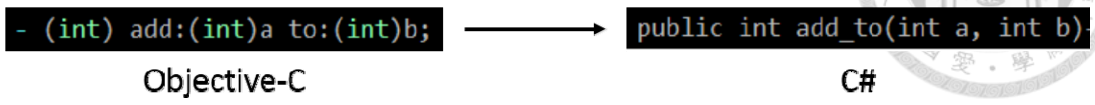


```
//Class.h
#import<Foundation/Foundation.h>
@interface MyClass : NSObject
{
 @public
   int i, j;
   float f;
 @private
   BOOL b;
}

- (int) add:(int)a to:(int)b;
@end
```

Figure 3.2 Function declarations with particular format

The Figure 3.2 shows this particular format in Objective-C. As a function name in C#, we decide to concatenate the separated part of the selector with underline.

Figure 3.3 shows the change between the original code in Objective-C and the C# code we generate.



```
- (int) add:(int)a to:(int)b;
```

```
public int add_to(int a, int b);
```

Figure 3.3 Translation of function declaration with particular format

d. Argument

The type of each argument should be processed correctly. To retain the original meaning, the name of argument should consist with the original name in the Objective-C code.

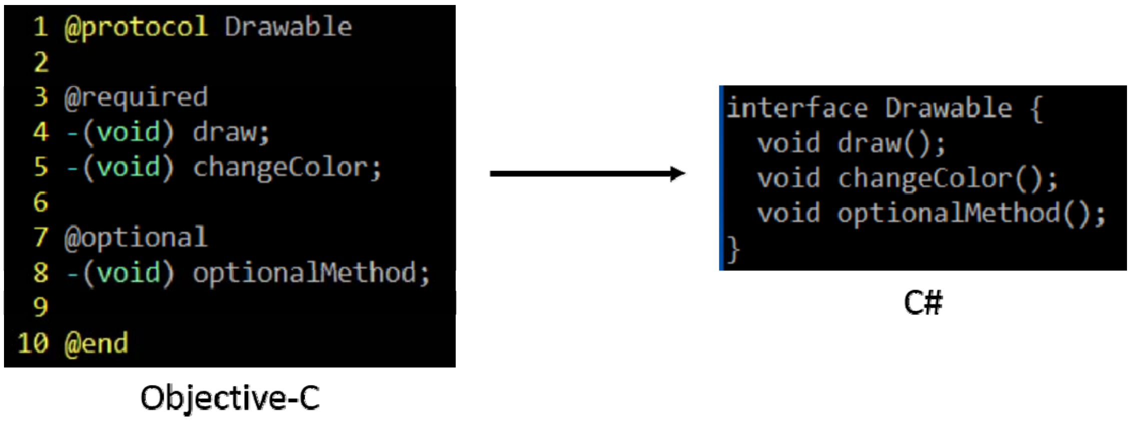
The relation of inheritance is handled in our tool to retain the object model designed in original program.

3.2 Protocol

Object-oriented language should support that define a set of behavior that is expected of an object in a given situation. In Objective-C, due to it just allows single inheritance; it allows you to define protocols, which declare the methods expected to be used for a particular situation.

A class interface declares the methods and properties associated with that class. A protocol, by contrast, is used to declare methods and properties that are independent of any specific class.

Interface in C# has the similar meaning with protocol in Objective-C, so we decide to translate this syntax in Objective-C into interface of C#. Figure 3.4 shows the format of protocol in Objective-C and the translation between Objective-C and C#.



```
@protocol Drawable

@required
-(void) draw;
-(void) changeColor;

@optional
-(void) optionalMethod;

@end
```

```
interface Drawable {
  void draw();
  void changeColor();
  void optionalMethod();
}
```

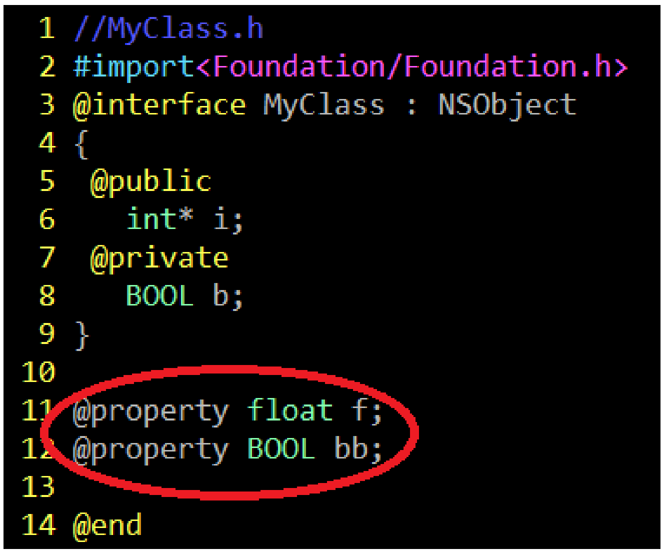
Figure 3.4 Translation of protocol

By default, all methods declared in protocol are required methods. This means all of the method declared in protocol should be implemented if the class conform the protocol. In Objective-C, it supports the optional methods. This means method declared in the optional section of protocol can be implemented if only required the method. Our tool does not support optional method which is in Objective-C, due to there is no similar meaning in C# with optional method which is in Objective-C. So, we take optional method as required method.

3.3 Property

In object-oriented programming, an object should hide its internal workings behind its public interface. It is important to access an object’s properties using behavior exposed by the object rather than trying to gain access to the internal values directly.

Objective-C properties offer a way to define the information that a class is intended to encapsulate. Property declarations are included in the interface section for a class; Figure 3.5 shows the format of property.

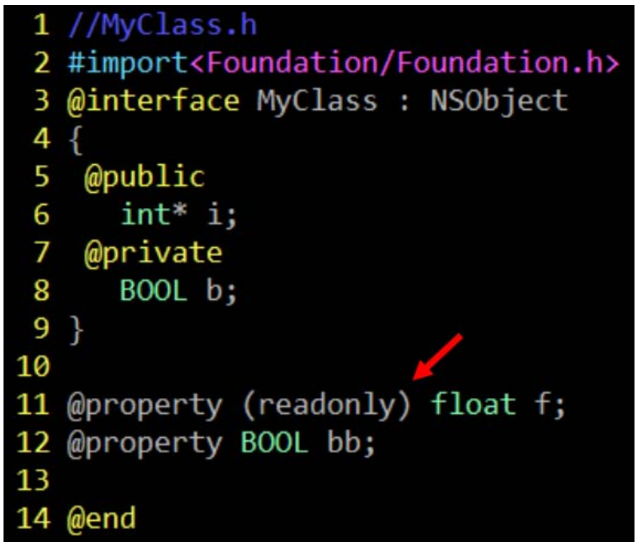


```
//MyClass.h
#import<Foundation/Foundation.h>
@interface MyClass : NSObject
{
 @public
   int* i;
 @private
   BOOL b;
}

@property float f;
@property BOOL bb;

@end
```

Figure 3.5 Property in Objective-C

If there is a property declaration in interface section, by default, these accessor methods are synthesized automatically by the compiler. The method used to access the value (the getter method) has the same name as the property. The method used to set value (the setter method) starts with the word "set" and then uses the capitalized property name.

Attribute of property describes some specific character of the property. Figure 3.6 shows the property adds the "readonly" attribute. It means that compiler should only generate the getter method.

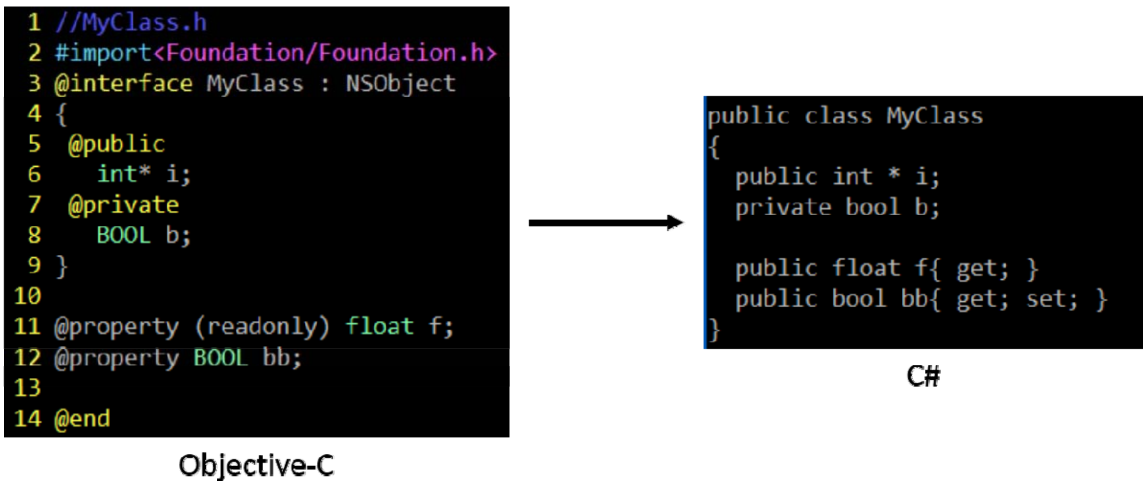


```
//MyClass.h
#import<Foundation/Foundation.h>
@interface MyClass : NSObject
{
 @public
   int* i;
 @private
   BOOL b;
}

@property (readonly) float f;
@property BOOL bb;

@end
```

Figure 3.6 Property with attribute in Objective-C

The opposite of readonly is readwrite which means compiler should generate both getter and setter method. There's no need to specify the readwrite attribute explicitly, because it is the default setting.

C# provides the syntax of similar meaning with property of Objective-C. Figure 3.7 shows the translation between property in Objective-C and get/set in C#.



```
//MyClass.h
#import<Foundation/Foundation.h>
@interface MyClass : NSObject
{
 @public
   int* i;
 @private
   BOOL b;
}

@property (readonly) float f;
@property BOOL bb;

@end
```

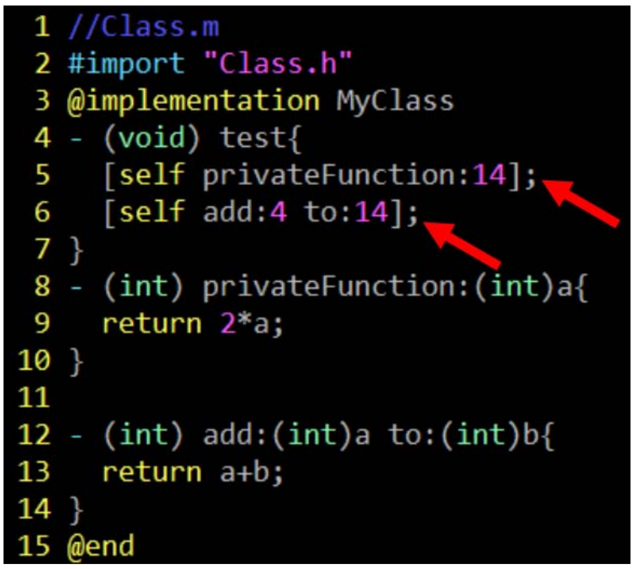
```
public class MyClass
{
  public int * i;
  private bool b;

  public float f{ get; }
  public bool bb{ get; set; }
}
```

Figure 3.7 Translation of property

In Objective-C, memory management is not automatic, programmer should handle allocating and deallocating of object, so property has some special attribute for memory management. These attribute is just ignored, cause of there is garbage collection in C#.

3.4 Message Passing

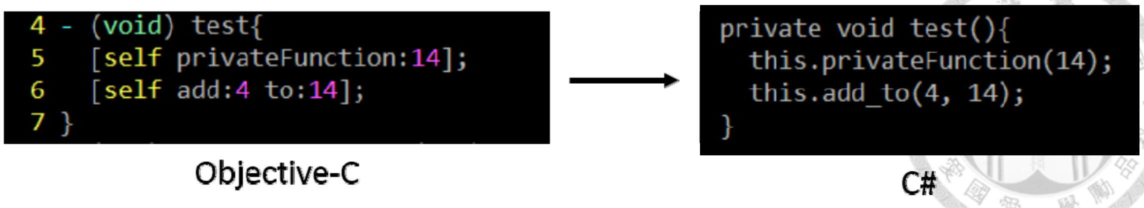
Message Passing is the way for interaction between object and object in Objective-C, in other words, method invocation, by contrast with the function call in C#.



```
//Class.m
#import "Class.h"
@implementation MyClass
- (void) test{
  [self privateFunction:14];
  [self add:4 to:14];
}
- (int) privateFunction:(int)a{
  return 2*a;
}

- (int) add:(int)a to:(int)b{
  return a+b;
}
@end
```

Figure 3.8 Message passing in Objective-C

Figure 3.8 sows the format of message passing in Objective-C. In message passing the first argument is called receiver, by contrast in using normal function call language like C#, callee. Except object name, receiver may be “self” or “super”. When the receiver is “self”, it means that the message is passing to itself. When the receiver is “super”, it means that the message is passing to the base class of itself. Rest part is the selector of function and arguments. To retain the original semantic meaning, we choose to translate message passing into Objective-C into function call in C#. Figure 3.9 shows the syntax of the function call in C# and the translation between Objective-C and C#.



```
- (void) test{
  [self privateFunction:14];
  [self add:4 to:14];
}
```

```
private void test(){
  this.privateFunction(14);
  this.add_to(4, 14);
}
```

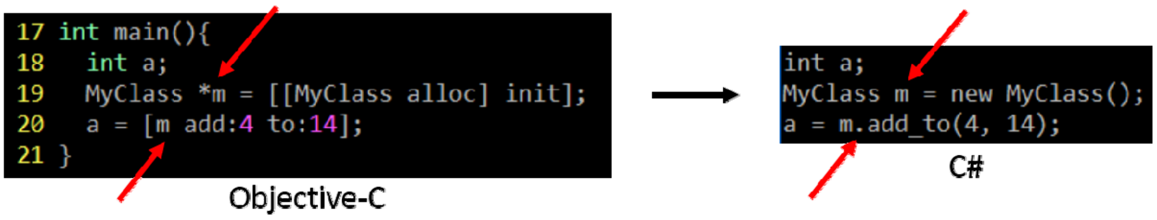
Figure 3.9 Translation between message passing and function call

For converting receiver to C#, “self” should be translated into “this”, and “super” should be translated into “base”. Selector in this part is processed as same as class definition, concatenating the separated part of the selector with underline.

3.5 Pointer

In Objective-C, using pointer with message passing is the basic way to make object interacts. Unfortunately, pointer in C# is just allowed pointing to primitive type or struct of primitive type. Almost of all object is reference in C#, and all of them are allocated in heap as same as Objective-C. So, we need to eliminate the use of pointer, and we handle the three major parts of the use of pointer.

1. The declaration of object pointer in Objective-C is translated into object reference and the operation of message passing should be translated into (.) operation. Figure 3.10 shows the result of translation.



```
int main(){
  int a;
  MyClass *m = [[MyClass alloc] init];
  a = [m add:4 to:14];
}
```

```
int a;
MyClass m = new MyClass();
a = m.add_to(4, 14);
```

Figure 3.10 Change of pointer processing

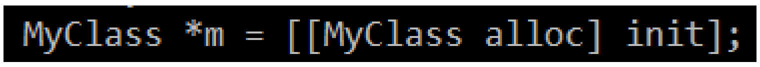
2. The method definition in class definition should also be modified. If the argument type of function is pointer type, it should be translated into reference.

3. The function with return type which is pointer should be translated; we handle

the construction of object which is the most common function in this situation.

3.6 Object Construction

In Objective-C, the construction of the object is separated into two operations. The one is for memory allocation; the other is the initialization of class members.

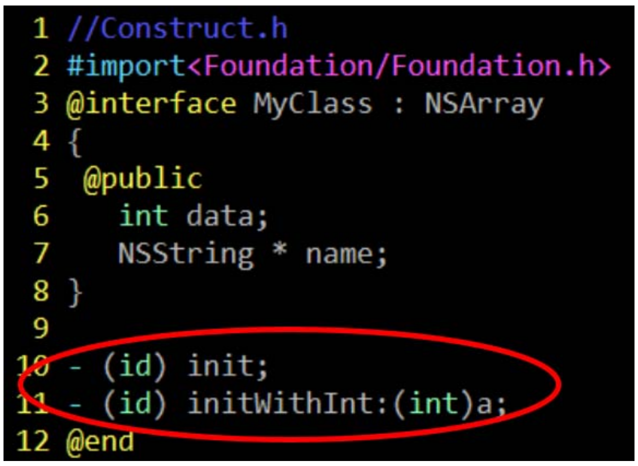


```
MyClass *m = [[MyClass alloc] init];
```

Figure 3.11 Construction of the object in Objective-C

Figure 3.11 shows the construction of the object in Objective-C. “Alloc” and “init” are two of the functions inherited from NSObject, the root object of the entire object in Objective-C. These two functions are always combining to use to construct the object in Objective-C, so we process the construction of object in two parts. The one is the definition of initialization function; the other is object construction in statement.

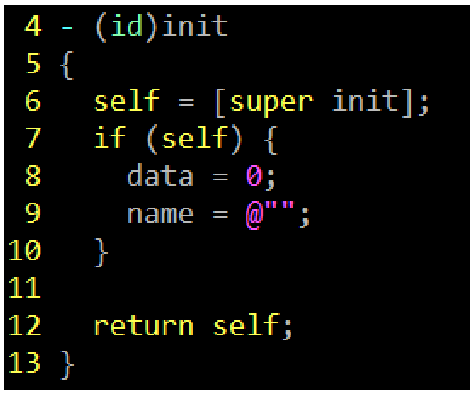
In definition of initialization function, we translate all of the initialization function into constructor in C#. Objective-C does not support function overloading, so there would be different name for function of initialization. Figure 3.12 shows this situation in Objective-C.



```
//Construct.h
#import<Foundation/Foundation.h>
@interface MyClass : NSArray
{
 @public
   int data;
   NSString * name;
}

- (id) init;
- (id) initWithInt:(int)a;
@end
```

Figure 3.12 Different name for function of initialization

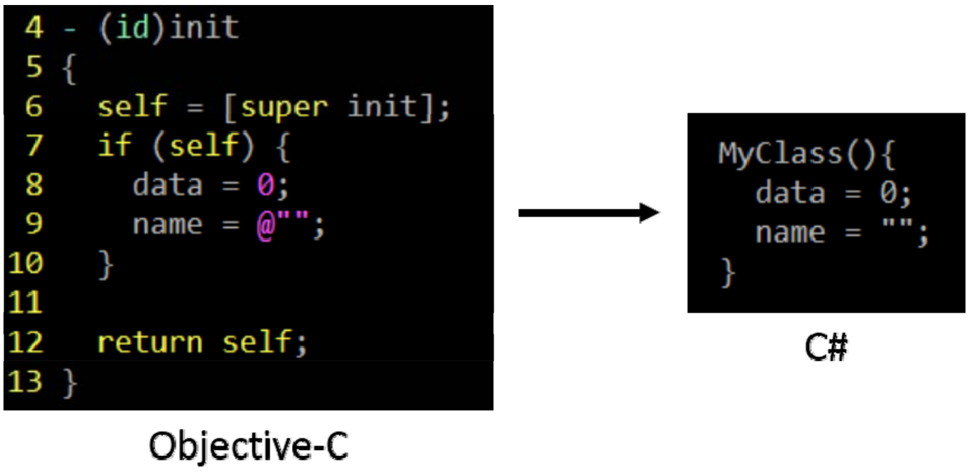
We take a function as an initialization function when it meets two conditions. The one is that the name of function is start with “init”; the other is that return type of the function is id or pointer of class. If a function meets these two conditions, it is regarded as initialization function in Objective-C, and we translate it into the format of constructor in C#.



```
- (id)init
{
  self = [super init];
  if (self) {
    data = 0;
    name = @"";
  }

  return self;
}
```

Figure 3.13 Common definition of initialization function in Objective-C

Figure 3.13 shows the common definition of initialization function in Objective-C. In this function of initialization, “self” is a pointer point to the object itself. There are several lines of code for checking if the initialization of super class is normal, and return self at the end. The translation eliminates these codes which is redundant in C#. Figure 3.14 shows the translated result.

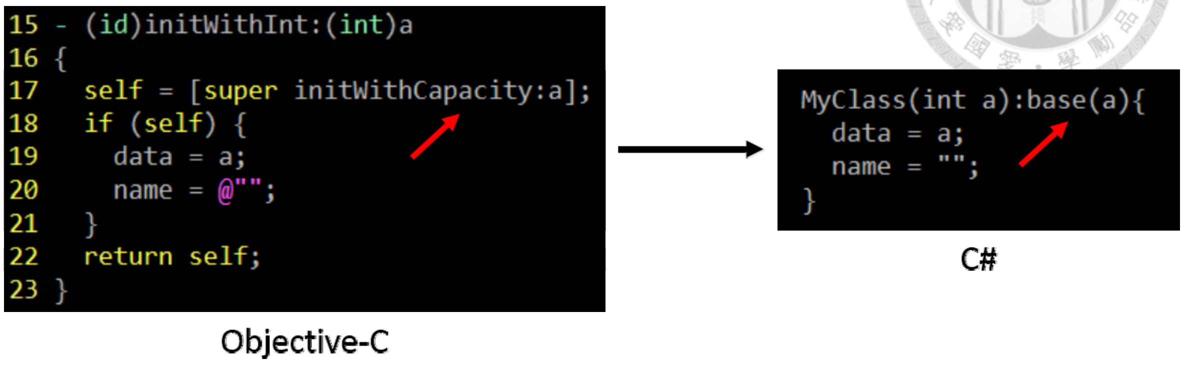


```
- (id)init
{
  self = [super init];
  if (self) {
    data = 0;
    name = @"";
  }

  return self;
}
```

```
MyClass(){
  data = 0;
  name = "";
}
```

Figure 3.14 Translation between initialization function and constructor

If the function of initialization of super class is not the default function, we add it on initialize list in C# with right argument. Figure 3.15 shows this situation.



```
- (id)initWithInt:(int)a
{
  self = [super initWithCapacity:a];
  if (self) {
    data = a;
    name = @"";
  }
  return self;
}
```

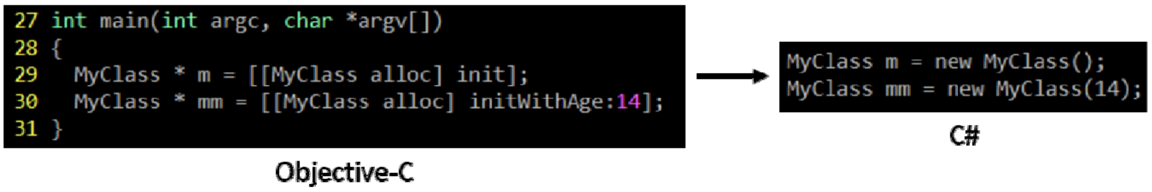
Objective-C

```
MyClass(int a):base(a){
  data = a;
  name = "";
}
```

C#

Figure 3.15 Initialization function with specific construct of super class

For handling the construction of object in the common statement of Objective-C, we translate the combination of “alloc” and initialization function into the calling of constructor in C#. Figure 3.16 shows the translation of statement between Objective-C and C#.



```
int main(int argc, char *argv[])
{
  MyClass * m = [[MyClass alloc] init];
  MyClass * mm = [[MyClass alloc] initWithAge:14];
}
```

Objective-C

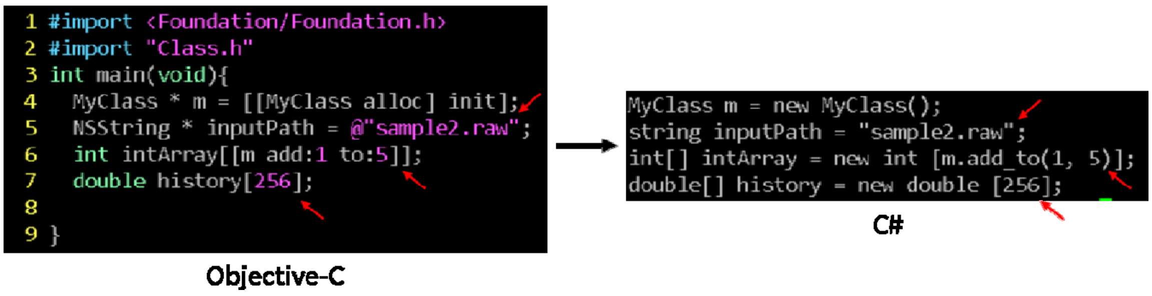
```
MyClass m = new MyClass();
MyClass mm = new MyClass(14);
```

C#

Figure 3.16 The statement of object construction

3.7 Array, string literal

There is some syntax difference between Objective-C and C#. Figure 3.17 shows the array definition and string literal in Objective-C and C#, we also translate these into right syntax in C#.



```
#import <Foundation/Foundation.h>
#import "Class.h"
int main(void){
  MyClass * m = [[MyClass alloc] init];
  NSString * inputPath = @"sample2.raw";
  int intArray[[m add:1 to:5]];
  double history[256];

}
```

```
MyClass m = new MyClass();
string inputPath = "sample2.raw";
int[] intArray = new int [m.add_to(1, 5)];
double[] history = new double [256];
```

Figure 3.17 Translation of array and string literal

Chapter 4 Design and Implementation

This section describes how our tool benefits from using Clang and provides the implementation details for the translator.

4.1 Overview of Translator

Our tool is implemented as a Clang plug-in in Clang 3.4 and LLVM 3.2. Clang plug-in provide a way to add extra steps to the original Clang compilation. Our translator is a source-to-source compiler that operates at the AST level. Parsing input source code to generate AST is the most essential task of many compilers. We chose to leverage Clang to design and implement the translator. Our tool uses Clang to parse and generate AST, focusing on the manipulation of the generated AST using a subclass of Clang "ASTConsumer." Our tool also uses Clang Rewriter, a Clang library, to perform string-based rewrite that inserts, removes, and substitute text to specific ranges of the input source code. The modified result is stored in another buffer, so that the original source and the AST are not altered.

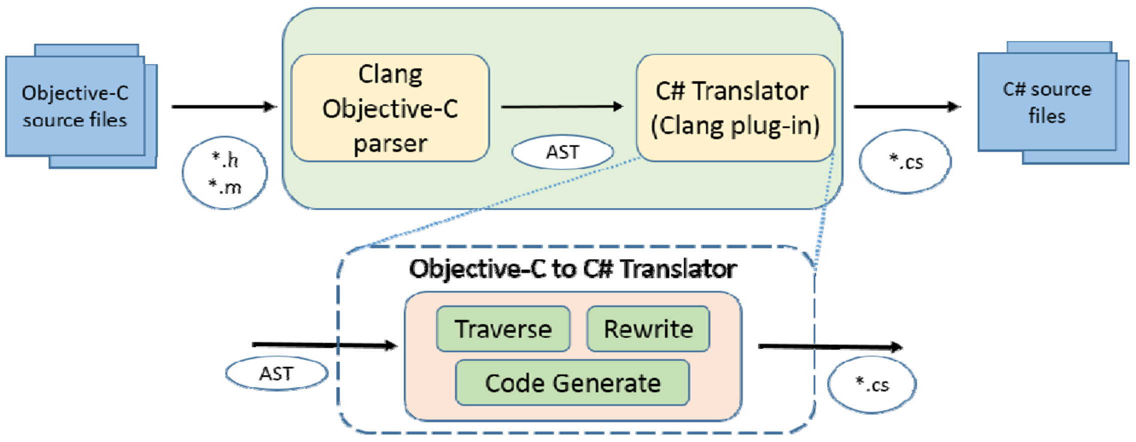


Figure 4.1 High level overview of the translator

Figure 4.1 shows the high level overview of the translator. The Objective-C to C# translator takes AST generated by Clang parser as input, then output the code in C#. The translator does three jobs on the AST: (1) traverse the AST node; (2) rewrite the code to

C# format and save it in buffer; (3) generate code in right way. They are recursively executed until traversing of the AST is finished.

4.2 Object Organization of the Translator

View from the angle of object organization, the translator consists of five generators. We get the root set of all the declaration in the function inherited from ASTConsumer, and determine each declaration is which kind of declaration. Then the declaration is handled by specific generator to generate code in C#. Figure 4.2 shows all kind of the generator and the design in our tool.

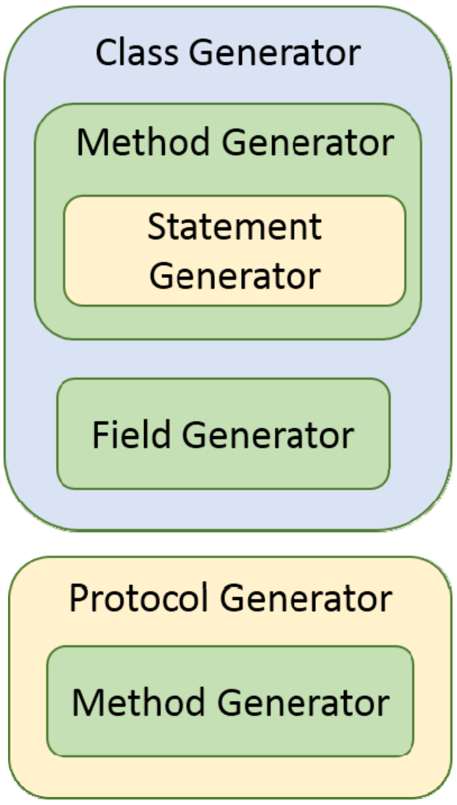


Figure 4.2 Object organization in the translator

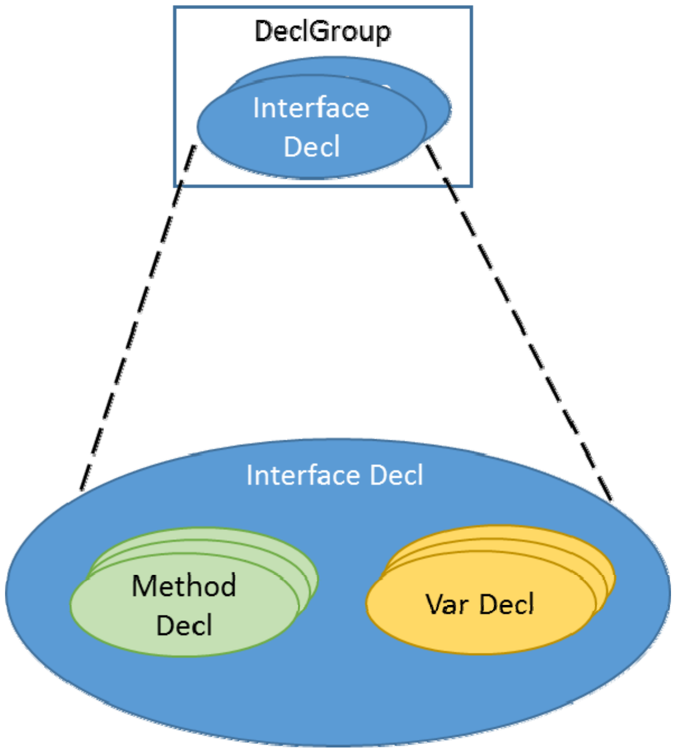


Figure 4.3 The interface declaration node in abstract syntax tree of Clang

If the interface declaration is determined, the translator calls class generator to handle and generate the code of class definition in C#. Figure 4.3 shows the interface declaration node in abstract syntax tree in Clang. To generate the code of whole class, class generator translates the class definition into C# format, and traverses all of the method declaration and variable declaration of interface declaration. Class generator delivers the method declaration to method generator to handle and generate the function part of the class, and delivers the variable declaration to field generator to handle and generate the member part of the class.

After method generator got the method declaration, it generates the method format in C#. Then method generator delivers the root set of statements to statement generator to handle the function body of the method.

Statement generator traverses all statements of the set passed by method generator, and generates each statement in C# format.

Protocol generator is called when the protocol declaration is determined in root set of declaration. It handles the translation of protocol we mentioned at chapter 3. The protocol declaration has the set of method declaration which is declared in protocol. Protocol generator passes these method declarations to method generator to generate the function declaration of these methods in C# format.

Chapter 5   Evaluation

In this part, we present the practical translation of two programs, and discuss and evaluate how well the translator performs.

5.1   Click and Count

Click and Count is a simple iOS application. It has three objects on the screen, a label, a text field, and a button. The text field shows an integer, and 0 is showed at the beginning. If the button is clicked, the integer on text field would plus 1 until the application eliminated. Figure 5.1 shows the snapshot of Click and Count.

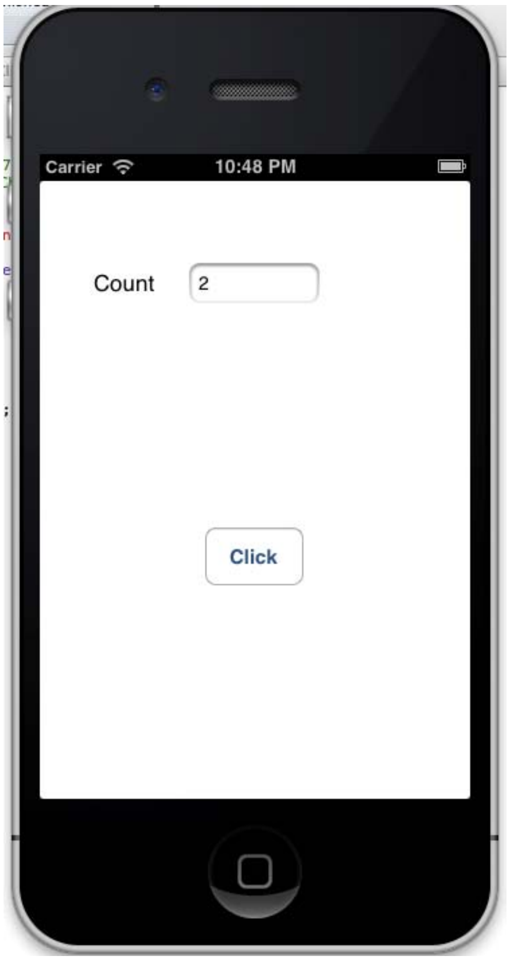


Figure 5.1 The snapshot of Click and Count

There is only one class in model part of Click and Count which is named ClickCount. ClickCount has an integer member for counting the time of pressing the button from the starting of this program. There are three interfaces in ClickCount: (1) plus the integer member; (2) get the number of the integer member; (3) get the number of the integer member as NSString.

The translation result for ClickCount is 14 lines of code in C#, and only one line need to be modified. The reason of modification is the difference in string between Cocoa framework and mono framework.

5.2 Outlining

Outlining is an application more complicated than Click and Count. There are two buttons at the bottom of the screen. If the left button is clicked, the middle of the screen would show the image in Figure 5.2. If the right button is clicked, the program would take the coins image as input, and process to outline the boundary of coins in Figure 5.3.

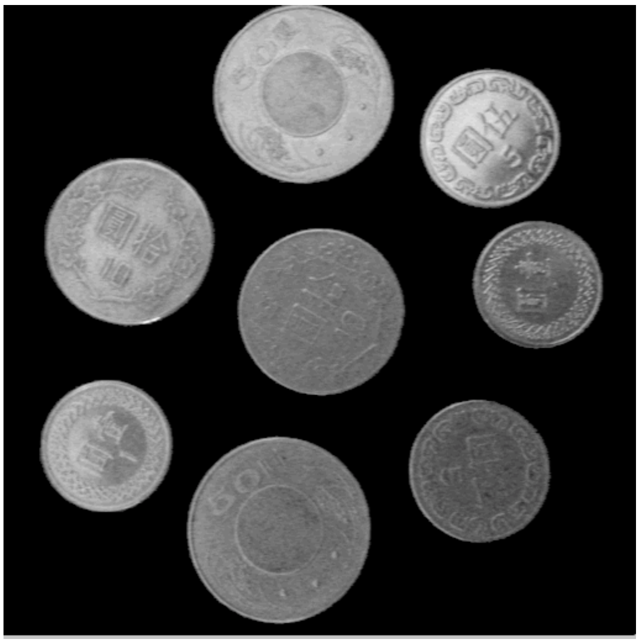


Figure 5.2 The coins image before Outlining process

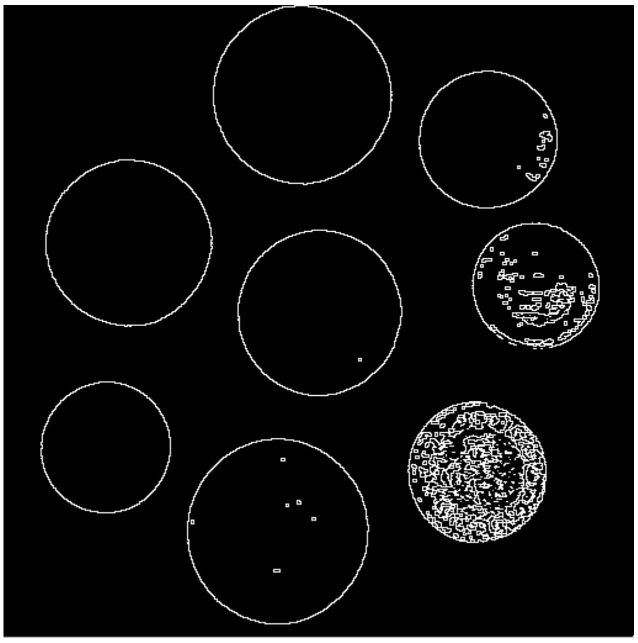


Figure 5.3 The coins image after Outlining process

There is also one class in model part of Outlining which is named Outline. Outline implements an algorithm to process the image for outlining the boundary. It has an interface to get the processed data. The control part of the Outlining calls Outline to get the processed data, and make the view to show the image on the screen.

The translation result for Outline is 120 lines of code in C#, and 8 lines of code need to be modified. The reason of modification is also the difference between Cocoa framework and mono framework.

Chapter 6 Conclusion and Future Work

The Objective-C to C# translator can be used to efficiently convert the object organization and the basic syntax in the Objective-C to the C#, which reduces routine workload of translation between the two programming languages. The Object-C to C# translator is available when the input follows some syntactic rules; programmers can benefit from this translator due to its function of reducing the number of lines in codes, and thus further aid programmers to save a vast amount of time on developing cross-platform.

Chapter 7 Reference

Lattner, C. (2008). LLVM and Clang: Next generation compiler technology. The BSD Conference.

Lattner, C. and V. Adve (2004). LLVM: A compilation framework for lifelong program analysis & transformation. Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE.

Xamarin http://xamarin.com/

Apple Developer http://developer.apple.com/

C# Programming Guide http://msdn.microsoft.com/en-us/library/

67ef8sbd(v=vs.90).aspx