САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Удалов Александр Николаевич

Прозрачное взаимодействие программ на языке Kotlin с библиотеками на Objective-C

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор А.Н. Терехов

подпись

Научный руководитель:
к. ф.-м. н. Д.Ю. Булычев

подпись

Рецензент:
А.А. Бреслав

подпись

Санкт-Петербург
2013

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Department of Software Engineering

Alexander Udalov

Transparent interoperability between Kotlin programs and Objective-C libraries

Graduation Thesis

Admitted for defence.
Head of the chair:
professor A. Terekhov

_____
signature

Scientific supervisor:
Ph. D. D. Boulytchev

_____
signature

Reviewer:
A. Breslav

_____
signature

Saint Petersburg
2013

Оглавление

Введение

Современным разработчикам знакома ситуация, при которой разные части сложного программного продукта приходится реализовывать на разных языках программирования. Обычно это связано с тем, что одна система лучше адаптирована под какую-то конкретную задачу, чем другая.

Один яркий пример такой ситуации связан с разработкой приложений под операционную систему Mac OS X. Разработчики приложений на платформе JVM (в частности, на языке Java) часто страдают от неудобства и неповоротливости стандартной библиотеки Swing [14], предназначенной для отрисовки элементов пользовательского интерфейса. Кроме того, приложения, написанные с использованием Swing, не выглядят «родными» для Mac OS. С другой стороны, существует удобный и более понятный способ создавать пользовательские интерфейсы под эту систему: Interface Builder, входящий в поставку Xcode [16], среды для разработки приложений под Mac OS. Но использовать интерфейсы, свёрстанные в Interface Builder из кода на JVM напрямую не представляется возможным, поскольку он создан и существует в другой экосистеме: приложения в Xcode разрабатываются на языке Objective-C.

Идеальным решением этой проблемы была бы возможность вызывать код библиотек на Objective-C из JVM, при этом корректно совершая преобразования данных на стыке двух систем. Обеспечить такую возможность — одна из целей данной работы. При этом, в отличие от существующих подобных проектов, интерфейс для доступа к коду на Objective-C предоставляется не для Java, а для другого языка, компилируемого для JVM — Kotlin [8].

Терминология

Под библиотекой на Objective-C в данной работе понимается динамически слинкованная библиотека (Dynamically Linked Library). Предполагается, что библиотека состоит из двух компонент:

- Машинный код библиотеки, двоичный файл с расширением .dylib. Этот файл нужен приложению, использующему библиотеку, на этапе исполнения, поскольку он содержит непосредственно инструкции, предназначенные для выполнения на процессоре.

- Заголовочные файлы (заголовки), набор текстовых файлов с расширением .h. Заголовки содержат код на языке C или Objective-C, перечисляющий декларации всех символов, экспортируемых библиотекой.

1. Постановка задачи

Цель данной работы — обеспечить прозрачное взаимодействие кода на языке Kotlin под платформу JVM с существующими библиотеками на Objective-C. Пользователь должен иметь доступ в коде на Kotlin к классам, методам и т.д., определенным в коде на Objective-C, и язык должен «видеть» эти декларации как свои, позволяя вызывать методы, создавать объекты и т.д. При этом код предполагается интегрировать в компилятор Kotlin для JVM, чтобы целиком избавить пользователя от использования сторонних утилит.

В рамках поставленной цели выделим несколько задач:

- Загрузка деклараций из заголовков библиотеки.

  Компилятору Kotlin необходимо каким-либо образом загрузить из заголовочных файлов код на Objective-C, провести лексический, синтаксический и семантический анализ и сохранить найденные декларации.

- Отображение деклараций в термины Kotlin.

  Компилятор должен не только знать о существовании подключенной библиотеки, но и уметь обращаться с внешними декларациями как с собственными. Для этого необходимо построить похожую иерархию классов, т.е. отобразить классы, протоколы, категории и другие понятия Objective-C в аналогичные понятия Kotlin. Также нужно преобразовать типы, предварительно создав необходимые типы, соответствующие объявленным в библиотеке. Преобразование типов, в частности, должно затрагивать типы возвращаемых значений и параметров методов библиотеки.

- Генерация JVM-классов.

  По построенной модели деклараций, классов, типов, присутствующих в библиотеке, компилятор должен сгенерировать байткод для корректного исполнения под JVM.

2. Обзор существующих решений

Поскольку Kotlin довольно молодой язык и не настолько популярный, как другие языки под платформу JVM, аналогичных по целям и результатам (взаимодействие Kotlin и Objective-C) проектов нет. Тем не менее, существует несколько проектов, обеспечивающих «мост» между JVM и Objective-C для языка Java, во многом схожем с Kotlin.

2.1. JNI

JNI (Java Native Interface) [7], вероятно, самый низкоуровневый способ обеспечить взаимодействие между JVM и Objective-C. JNI — это механизм, встроенный в JVM с первой версии, позволяющий байткоду JVM вызывать функции из библиотек, написанных на нативных для платформы языках, таких как C, C++, ассемблер и др. Поскольку Objective-C является надмножеством C и компилируется в нативный код, с помощью JNI и Objective-C Runtime, встроенной библиотеки, обеспечивающей рефлексию на этапе исполнения, возможно всё то же, что возможно и из самого Objective-C: отправка сообщений, получение класса у объекта, добавление новых классов в систему и т.д.

Однако, во-первых, пользователю необходимо написать некоторое количество кода на нативном языке (реализующего интерфейс, сгенерированный утилитой `javah` из JNI). Получающийся при этом код абсолютно не объектно-ориентированный и тяжёлый для понимания и поддержки. Во-вторых, пользователь должен сам заботиться о маршалинге типов[1]. Это непрактично, поскольку в большинстве случаев система заставляет пользователя писать одинаковый код раз за разом.

Тем не менее, накладные расходы при правильном использовании JNI довольно незначительны. Также JNI является единственным способом взаимодействия JVM с нативным кодом, и поэтому все остальные

[1]К примеру, если нативная функция, которую требуется вызвать из JVM, принимает параметром строку в виде `char *`, программист должен сам обеспечить корректное преобразование строки JVM (`java.lang.String`) в `char *`, а именно: сперва получить массив символов функцией `GetStringUTFChars` из JNI, затем проверить, не было ли брошено исключение в процессе (например, `OutOfMemoryError`) и затем освободить полученный массив функцией `ReleaseStringUTFChars`.

проекты прямо или косвенно основаны на JNI.

2.2. JNA

JNA (Java Native Access) [6] — библиотека для JVM, предоставляющая более лёгкий доступ к нативному коду, чем JNI. Главной целью при разработке JNA была прозрачность взаимодействия между JVM и нативным кодом. В частности, в отличие от JNI, пользователю вовсе не нужно писать нативного кода. Для того, чтобы иметь возможность вызвать функции нативной библиотеки из JVM, пользователю необходимо определить Java-интерфейс и перечислить в нём декларации тех функций, которые он собирается использовать. Типы параметров и возвращаемых значений в таком интерфейсе нужно указывать в соответствии с таблицей из документации JNA [4], и при правильном использовании конвертацию типов библиотека обеспечит сама. К примеру, C-функция, принимающая параметром строку типа `const char *`, в интерфейсе может быть обозначена методом, принимающим параметром объект типа `java.lang.String`.

Один из главных недостатков JNA в том, что, вероятно из-за соображений совместимости, библиотека не использует возможности последних версий Java (начиная с 5.0), такие, как аннотации, generic-типы и др. Это проявляется в неудобстве и недостаточной гибкости системы.

Ещё один недостаток JNA — для каждой используемой нативной функции необходимо *вручную* написать в коде на Java её декларацию. Эта проблема частично решена проектом JNAerator [5], который по заголовочным файлам нативной библиотеки генерирует код на Java, подходящий для использования с JNA. Однако в JNA для некоторых C-типов существует несколько корректных вариантов отображения в Java[2], поэтому пользователю часто приходится, снова вручную, исправлять уже сгенерированный код, адаптируя его под свои нужды.

Наконец, главным недостатком JNA в рамках поставленных задач является отсутствие поддержки Objective-C. При исполь-

[2]К примеру, нативному типу `const char *` можно сопоставить `String`, `Pointer`, `ByteByReference`, `ByteBuffer` или `byte[]`.

зовании JNA доступны лишь низкоуровневые нативные методы из Objective-C Runtime, обеспечивающие корректную диспетчеризацию сообщений, создание классов и т.д.: например, `objc_msgSend` или `object_getClassName`. Для действительно удобной работы с Objective-C-классами в JVM, пользователю, грубо говоря, необходимо самому определить каждый нужный класс, и в каждом методе вызвать `objc_msgSend`, корректно преобразовав аргументы и результат.

2.3. BridJ

BridJ [1] — молодой аналог JNA, целью которого является обеспечить прозрачное взаимодействие JVM со всем семейством C-языков: C, C++, Objective-C.

Основными целями при разработке BridJ являются простота использования и производительность. Проекту чуть более двух лет, и ему удалось избежать многих недостатков JNA. В частности, выразительность используемой типовой системы позволяет каждому нативному типу однозначно сопоставить тип в JVM[3]. По производительности BridJ выигрывает у JNA в большинстве случаев, а в некоторых случаях на порядки (например, при использовании `struct`).

Однако есть и недостатки:

- Как и с JNA, пользователю необходимо вручную по заголовочным файлам выписать декларации используемых функций, или сгенерировать их при помощи JNAerator и проверить.

- Objective-C теоретически поддерживается, но на деле, как показал анализ исходных кодов библиотеки, мало реализовано и протестировано. Кроме того, поддержка Objective-C не является приоритетной задачей этого проекта на данный момент.

- Поскольку проект молодой, некоторые возможности, присутствующие в JNA, пока не реализованы вовсе: к примеру, передача

[3]Возвращаясь к примеру с `const char *`, в BridJ ему будет соответствовать `Pointer<Byte>`, который при желании возможно привести к `String` или другому типу.

`struct` по значению как аргумента или результата функции.

2.4. Rococoa

Rococoa [13] — проект с наиболее полной поддержкой взаимодействия JVM и Objective-C. Это библиотека, при наличии которой возможно не только создавать Objective-C-объекты в JVM и вызывать их методы и свойства, но и реализовывать интерфейсы, определённые в Objective-C, на JVM. Как и BridJ, Rococoa в стадии разработки, на которой ещё не ясно, каков будет окончательный интерфейс библиотеки. Тем не менее, сейчас это, вероятно, самая популярная библиотека для интеграции проектов под JVM и Objective-C.

Rococoa основана на JNA в своей реализации, а значит, перенимает многие её достоинства и недостатки. Как и в остальных проектах, сперва по заголовкам необходимо сгенерировать интерфейсный код на Java, реализацию которого Rococoa предоставит на этапе исполнения.

Поддержка основных и часто используемых Objective-C-классов в Rococoa хорошо протестирована. Кроме того, в официальном репозитории библиотеки есть небольшие, но не тривиальные программы: например, одна из них, полностью написанная на Java, использует возможности системы нотификаций Growl [3] для отображения всплывающего окошка.

2.5. Выводы

Существует несколько библиотек, занимающихся созданием моста между Java и Objective-C. Один общий недостаток вышеописанных библиотек — для того, чтобы использовать внешние декларации, необходимо предварительно объявить их вручную в коде, правильно указав имена и типы. Другой общий недостаток в рамках поставленной задачи связан с тем, что в первую очередь эти средства ориентированы на язык Java: используя возможности компилятора Kotlin в байткод JVM несложно добиться работы этих средств в коде на Kotlin, но получающийся код не так выразителен, как мог бы быть.

3. Архитектура

Работа состоит из нескольких модулей в различных подсистемах компилятора языка Kotlin в байткод JVM.

- Front-end компилятора
  - Загрузка деклараций из заголовков
  - Преобразование иерархии классов
  - Преобразование типов
- Библиотека Kotlin Runtime
  - Kotlin Native
- Back-end компилятора
  - Генерация JVM-классов

3.1. Преобразование иерархии классов

Несмотря на то, что и Kotlin, и Objective-C — объектно-ориентированные языки, в их моделях наследования есть некоторые различия. Цель этой части — преобразовать полученные объекты, соответствующие декларациям из заголовков Objective-C, так, чтобы компилятор Kotlin «видел» эти декларации как собственные.

Все декларации верхнего уровня, полученные из заголовков, в Kotlin попадают в пакет под названием `objc`. Таким образом, к примеру, чтобы использовать класс `NSObject` из библиотеки Foundation Kit [2] по простому имени, в коде на Kotlin возможно заимпортировать этот класс строкой `import objc.NSObject`.

3.1.1. Классы

Класс Objective-C (объявленный ключевым словом `@interface`) представляется классом (`class`) с таким же именем в Kotlin. Види-

мость такого класса в Kotlin — `public`, поскольку заголовочные файлы на деле содержат именно то, что должно быть видно пользователю интерфейса снаружи. Модальность[4] — всегда `open`, потому что в Objective-C не бывает абстрактных (т.е. реализованных частично), равно как и ненаследуемых классов. Наследование классов сохраняется: к примеру, классу `NSValue` из Foundation Kit, который наследуется от `NSObject`, в Kotlin будет соответствовать класс `NSValue` с базовым классом `NSObject`. Список протоколов, реализуемых классом, абсолютно аналогично отображается в список супертипов класса в Kotlin.

3.1.2. Протоколы

Протокол (`@protocol`) представляется `trait`'ом в Kotlin. Видимость, аналогично классам, `public`, по тем же причинам. Модальность — `abstract`, как и у любого `trait`'а в Kotlin.

Однако имя `trait`'а не всегда совпадает с именем протокола, поскольку в Objective-C классы и протоколы существуют в разных пространствах имён и поэтому совпадения имён возможны, в отличие от Kotlin, где классы и `trait`'ы из одного пакета обязаны иметь разные имена. Было рассмотрено два решения этой проблемы:

1. Создать специальный отдельный пакет для `trait`'ов, полученных из протоколов Objective-C: скажем, `objc.protocols`. Тогда, поскольку все классы и протоколы оказались в разных пакетах, их имена могут совпадать. К примеру, полное имя класса `NSObject` из Foundation окажется `objc.NSObject`, а полное имя протокола `NSObject` — `objc.protocols.NSObject`.

2. Каждый протокол, совпадающий по имени с некоторым классом, отобразить в `trait` с похожим, но другим именем. Например, протоколу `NSObject`, имя которого совпадает с существующим классом, будет соответствовать `trait` с именем `NSObjectProtocol`[5].

[4] В терминологии Kotlin одно из трёх значений `open`, `final` или `abstract`, описывающее способность класса к наследованию и созданию экземпляров.

[5] При этом, разумеется, в маловероятном случае, при котором существует класс или протокол по

Неудобство первого решения в том, что по коду, в котором встречается имя некоторого класса, не заглядывая в заголовок файла с `import`-директивами, пользователь не может определить, имеется ли в виду здесь класс или протокол: в Objective-C такой проблемы не возникает, поскольку класс и протокол всегда находятся в разных синтаксических позициях. Поэтому было решено остановиться на втором варианте.

Список базовых протоколов каждого протокола аналогично классу отображается в список супертипов этого `trait`'а в Kotlin.

3.1.3. Категории

Категория в Objective-C — конструкция, позволяющая добавлять методы к классам вне их определения. Каждой категории соответствует `public trait` с таким же именем в Kotlin, который становится супертипом класса, интерфейс которого дополняет категория. К примеру, по категории `NSStringExtensionMethods` к классу `NSString` из Foundation будет создан `trait NSString+NSStringExtensionMethods`[6], и класс `NSString` в Kotlin унаследует этот `trait`.

Идейно методы в категориях сопоставимы с `extension`-методами Kotlin, однако было решено не производить именно такое преобразование (категория $\mapsto$ группа `extension`-методов) по следующим причинам:

1. Метод категории некоторого класса в Objective-C имеет приоритет над методом с аналогичным названием, определённым в интерфейсе класса, в то время как в Kotlin ситуация прямо противоположная. Это может спровоцировать проблемы на этапе разрешения вызовов Objective-C-методов.

2. Категории используются для группирования дополняющих методов, имеющих схожую направленность. В частности, у каждой публичной категории есть имя, которое было бы потеряно при таком преобразовании.

имени `NSObjectProtocol`, предпринимаются дальнейшие попытки обеспечить уникальности имени: `NSObjectProtocol1`, `NSObjectProtocol2` и т.д.

[6]Имя через «+» — конвенция обозначения категорий и именования соответствующих файлов в Objective-C

3.1.4. Методы

Методы в Objective-C делятся на два вида: методы класса и методы экземпляра. Методам экземпляра, определённым в классах, соответствуют методы (с модальностью `open`) классов Kotlin. Методам экземпляра, определённым в протоколах и категориях соответствуют абстрактные методы `trait`'ов Kotlin. Методы же класса представляются методами в метаклассе и `class object` соответствующего контейнера (см. 3.1.5).

Концепции имён методов в Objective-C и Kotlin отличаются: имя метода в Objective-C состоит из нескольких частей, каждая из которых синтаксически находится перед одним из параметров этого метода и должна находиться перед аргументом, передаваемым методу. В Kotlin имя метода, как и в большинстве других объектно-ориентированных языков, состоит из одного идентификатора, и точно преобразовать имя Objective-C-метода, кроме как соединить все части с некоторым разделителем, не представляется возможным. Однако поскольку в большинстве[7] случаев первая часть имени метода прекрасно описывает действие, им производимое, было принято решение оставлять от имени метода только первую часть, если это не приводит к неоднозначности. К примеру, методу `- (id)initWithBytesNoCopy:length:encoding:freeWhenDone:` из `NSString` соответствует метод по имени `initWithBytesNoCopy` в классе `NSString` в Kotlin. Если первая часть имени метода всё-таки неоднозначно идентифицирует его, то имя будет образовано соединением всех частей через разделитель. Например, в том же классе в Kotlin существуют методы `initWithFormat`, `initWithFormat$arguments`, `initWithFormat$locale` и т.д., поскольку первая часть их имени неуникальна.

Каждому параметру Objective-C-метода соответствует параметр с тем же именем в Kotlin. Типы каждого из параметров и возвращаемого значения метода преобразуются в соответствии с 3.2.

[7] К такому выводу привёл анализ стандартных Objective-C-библиотек Foundation и Cocoa

3.1.5. Метаклассы

Метакласс Objective-C хоть и не является отдельной синтаксической конструкцией, но заслуживает особого внимания. Метакласс класса X в Objective-C — такой класс, у которого есть единственный экземпляр, объект (так же называемый `class object`) по имени X. Этот объект ведёт себя как остальные объекты в программе, в частности ему можно посылать сообщения и получать результат. Методы, соответствующие сообщениям, на которые отвечает такой объект, определены в самом классе X (со знаком «+» перед именем).

Особый случай, связанный с метаклассами, заключается в том, что если класс Y наследуется от класса X, то метакласс Y наследуется от метакласса X [10]. При этом, поскольку существуют метаклассы протоколов, которые поддерживают множественное наследование, метакласс на самом деле может иметь более одного класса в списке супертипов. В Kotlin существует такое понятие, как `class object`, но его тип не наследуется от типа `class object` базового класса. Этот факт не позволяет «бесплатно» сопоставить иерархии двух языков.

Кроме того, метакласс верхнего в иерархии класса (в Foundation это обычно `NSObject`) наследуется от этого класса. Это позволяет использовать экземпляры метаклассов как обычные объекты, на которых доступны все стандартные методы класса `NSObject`.

Проблема была решена следующим образом: по каждому классу X из Objective-C в Kotlin строится, помимо самого класса X, `trait` `<metaclass-for-`X`>`[8] и `class object`, наследующий этот `trait` и самый верхний предок X. Методы метакласса попадают в `<metaclass-for-`X`>`. В отличие от `class object`, `trait`'ы могут наследоваться, в том числе множественно. Кроме того, каждый `class object`, соответствующий экземпляру метакласса, наследуется от верхнего в иерархии класса, следовательно, такая конструкция в точности соответствует концепции метаклассов Objective-C. Пример построенной иерархии для двух классов из Foundation, `NSObject` и `NSValue`, изображён на рис. 1.

[8]Имена такого вида существуют в компиляторе Kotlin лишь для абстракции обозначаемой сущности; при генерации байткода они заменяются на имена, подходящие для JVM.

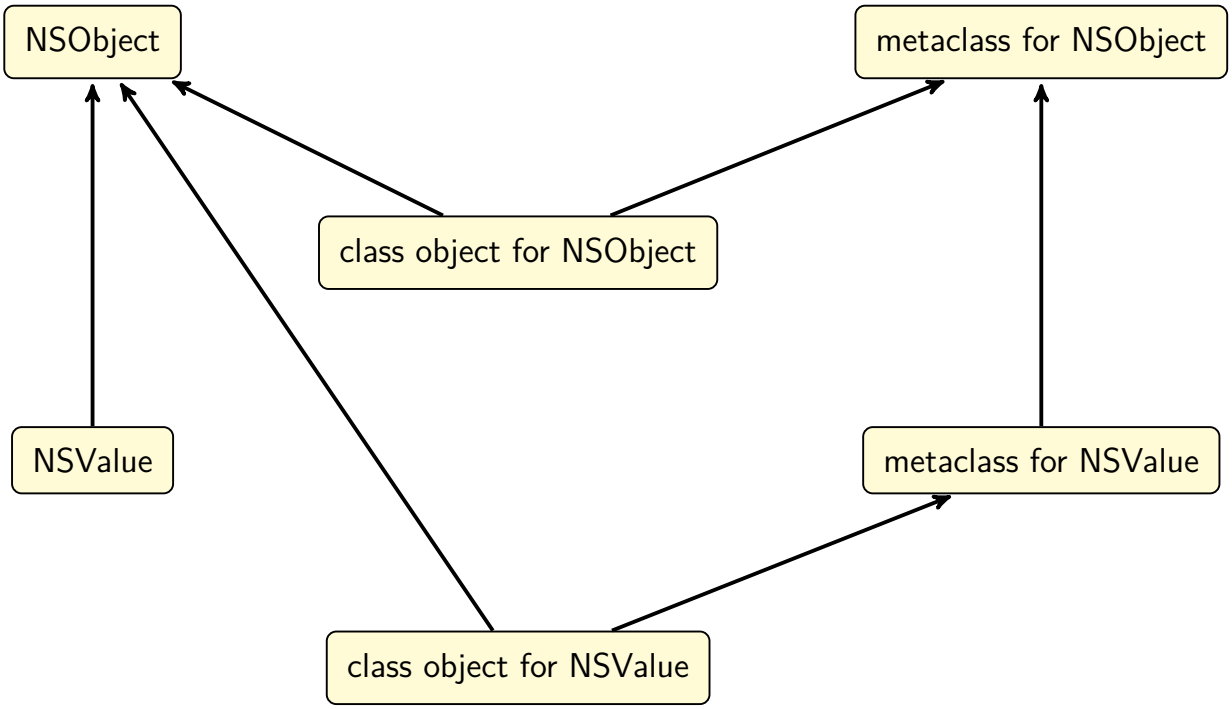

Рис. 1: Пример построенной иерархии с метаклассами на основе классов `NSObject` и `NSValue`

У протоколов и категорий также могут быть методы класса, поэтому для каждого протокола и категории создаётся `trait`, описывающий метакласс. Но у протоколов нет `class object`, а у категории `class object` совпадает с таковым у класса, который дополняет категория. Поэтому с протоколами и категориями ситуация слегка проще: `class object` для них не создаётся.

3.1.6. Свойства

Свойствам (`@property`) сопоставляются свойства (properties) Kotlin. При этом наличие атрибутов `readonly` или `readwrite` определяет, изменяемо ли свойство (`val` и `var` соответственно). Тип свойства преобразуется в соответствии с 3.2.

Поскольку с введением свойств в Objective-C 2.0 практически отпала необходимость доступа извне к переменным объекта (instance variables), они в данной работе не поддерживаются.

3.1.7. C-декларации

Так как Objective-C — надмножество языка C, в нём поддерживаются все и используются некоторые конструкции этого языка.

Тип-перечисление (`enum`) отображается в целочисленный тип `Int` в Kotlin. Каждой из констант перечислимого типа соответствует целочисленная константа с тем же именем в Kotlin.

Надо заметить, что, несмотря на семантическую схожесть `enum` с `enum class` в Kotlin, есть существенное отличие, не позволяющее сделать именно такое преобразование: значениями этого типа в C могут быть не только сами константы, но и побитовое «ИЛИ» любого подмножества множества этих констант, в то время как в Kotlin значением перечислимого типа может быть только одна из предопределённых констант. Кроме того, в стандартной библиотеке Cocoa распространены *безымянные* перечисления, представляющие из себя лишь список числовых констант, в комментарии к которым указано, где должны быть использованы значения этих констант. Подобные типы фактически невозможно разумно отобразить в `enum class`, поскольку у них нет имени.

Объединение (`union`) не поддерживается в рамках данной работы, поскольку эта конструкция слишком низкоуровневая, она не используется в публичных интерфейсах библиотек Cocoa и Foundation и ей нет аналогов в Kotlin.

Структура (`struct`) не поддерживается в рамках данной работы по причине сложности реализации.

3.2. Преобразование типов

3.2.1. Примитивные типы

Правила преобразования примитивных типов перечислены в таблице 1. Для каждого базового типа Objective-C был выбран примитивный тип Kotlin, наиболее точно отражающий его суть и имеющий размерность в байтах не меньше, чем его нативный аналог. Размерности базо-

вых типов в Mac OS X соответствуют таблице 3.1 из System V ABI [15].

Тип в Objective-C	**Тип в Kotlin**
`void`	`jet.Unit`
`int` `unsigned int`	`jet.Int`
`short` `unsigned short`	`jet.Short`

`long`
`unsigned long`
`long long`
`unsigned long long`
	`jet.Long`

`char` `unsigned char`	`jet.Char`
`float`	`jet.Float`
`double` `long double`	`jet.Double`
`BOOL`	`jet.Boolean`

Таблица 1: Правила преобразования примитивных типов

Некоторые тонкости такого преобразования:

- В Kotlin, как и вообще в JVM, отсутствуют беззнаковые типы. Однако знаковость типа при отображении игнорируется: например, `unsigned int`, равно как и `signed int`, становится `jet.Int`. Это сделано из соображений производительности и здравого смысла: чаще всего значения сущностей беззнакового типа, используемых в интерфейсах библиотек, не превышают границы допустимых значений аналогичного знакового типа: например, значения длин массивов, обычно представленные типом `size_t`, являющимся псевдонимом `unsigned int`, редко превосходят $2^{31}-1$ и, следовательно, могут быть проинтерпретированы как `signed int`. А в тех случаях, когда пользователю необходимы действительно большие

значения, ему подойдёт целочисленный тип большей размерности. В любом случае, поскольку значение в Kotlin представлено тем же количеством бит, при необходимости пользователь может самостоятельно получить настоящее значение беззнаковой величины в нужном ему виде.

- Символьные типы `char` и `unsigned char` представлены одним байтом в нативном коде, в то время как `jet.Char` отображается в JVM в примитивный тип `char`, размерность которого — два байта. Таким образом, при передаче аргумента типа `jet.Char` в нативный метод от него оставляется только младший байт. Более корректным решением было бы отображение `char` в тип `jet.Byte` размерности один байт, но такой подход менее оправдан семантически, к тому же пользователю пришлось бы вручную конвертировать каждый символ в байт в коде на Kotlin.

- 16-байтный тип чисел с плавающей точкой `long double` не поддерживается Objective-C[9] и на этапе исполнения представляется 8-байтным типом `double`. Поэтому, как и `double`, ему соответствует тип `jet.Double` в Kotlin.

3.2.2. Встроенные в Objective-C типы

1. `id` — тип в Objective-C, значение которого является указателем на объект неизвестного (нераскрытого) класса. `id` отображается в Kotlin в абстрактный класс `ObjCObject`, созданный для этой цели в библиотеке Kotlin Runtime, доступной каждому приложению на Kotlin на этапе исполнения. Отображённые верхние в иерархии классы наследуют `ObjCObject`, позволяя пользователю совершать неявные преобразования вверх по иерархии: к примеру, возможно передать методу аргумент типа `NSString`, если соответствующий параметр типа `id`. `ObjCObject` — абстрактный, потому что невозможно создать экземпляр типа `id`: для создания объекта нужно

[9] Согласно заметке к таблице 6.1 из [11].

инстанциировать конкретный класс.

2. `Class` — тип, значениями которого являются `class object`, указатели на классы. `Class` отображается в маркер-интерфейс `ObjCClass`, также добавленный в Kotlin Runtime, от которого наследуется корневой метакласс (метакласс верхнего в иерархии класса) и, следовательно, транзитивно наследуются все `trait`'ы, соответствующие метаклассам.

3. `SEL` — тип, значение которого является указателем на селектор (имя метода), отображается в класс `ObjCSelector`, который наследуется от `Pointer<Any>` (см. 3.2.6).

3.2.3. Указатели на объекты

В Objective-C возможно разместить объект класса только в динамической памяти (в отличие, например, от `struct`, которую можно разместить на стеке), следовательно, декларация не может иметь в качестве типа Objective-C класс: это потребовало бы размещения объекта на стеке при обращении к этой сущности. Типы, соответствующие *указателям* на объекты, отображены в соответствующие классы в Kotlin. К примеру, типу `NSObject *` в Objective-C соответствует тип `NSObject` в Kotlin.

3.2.4. Классы со списком реализуемых протоколов

Список протоколов, реализуемых классом, синтаксически находящийся в типе указателя на объект между именем класса и «`*`», участвует в отображении типа указателя. Результат отображения типа со списком протоколов — тип-пересечение (intersection type) типа класса и всех типов `trait`'ов, соответствующих протоколам. Если конкретный класс нераскрыт (в коде написано `id`), то пересечение производится с типом `ObjCObject` (согласно 3.2.2). При этом декларации, в которой встречается тип-пересечение, по требованию JVM добавляется типовой параметр с соответствующими ограничениями.

Примеры отображения некоторых деклараций типа указатель на объект со списком протоколов и без, приведены в таблице 2.

Декларация в Objective-C	**Результат в Kotlin**
`- (id)f`	`fun f(): ObjCObject`
`- (NSString *)f`	`fun f(): NSString`
`- (id <NSObject>)f`	`fun <T : ObjCObject> f(): T` `  where T : NSObjectProtocol`
`- (int)f: (NSValue <NSCoding> *)x`	`fun <T : NSValue> f(x: T): Int` `  where T : NSCoding`
`- (NSObject <NSCoding, NSCopying> *)f`	`fun <T : NSObject> f(): T` `  where T : NSCoding, T : NSCopying`
`@property NSValue <NSCoding> *f`	`val <T : NSValue> f: T` `  where T : NSCoding`

Таблица 2: Примеры преобразования типов-указателей на объекты[10]

3.2.5. Указатели на функции, блоки

Указатель на функцию с N параметрами отображается в функциональный тип `Function`N[11] в Kotlin. Типы параметров и возвращаемого значения функции обрабатываются рекурсивно.

Указатели на блоки (функциональные литералы в Objective-C) семантически эквивалентны указателям на функции и поэтому отображаются аналогично. Несколько примеров приведены в таблице 3.

3.2.6. Остальные указатели

Для случая произвольного указателя в Kotlin Runtime был создан специальный класс `Pointer<T>`. У этого класса один типовой параметр, который обозначает тип (отображённый рекурсивно), на который указывает указатель. Объект класса `Pointer` позволяет выделять и осво-

[10]Исходный код деклараций на Kotlin здесь приведён для наглядности. На деле, в реализации данной работы в коде компилятора Kotlin необходимые сущности строятся напрямую, минуя фазу синтаксического разбора.

[11]Один из типов `Function0<R>`, `Function1<R, P0>`, `Function2<R, P0, P1>`, ... Первый типовой параметр обозначает тип возвращаемого значения функционального объекта, а остальные N — типы параметров-значений.

Тип в Objective-C	**Тип в Kotlin**
`void (*)()`	`() -> Unit`
`void (^)()`	`() -> Unit`
`id (*)(double, char)`	`(Double, Char) -> ObjCObject`
`int (*)(int (*)())`	`(() -> Int) -> Int`

Таблица 3: Примеры преобразования указателей на функции

бождать нативную память, а также читать и записывать данные по указанному адресу.

Для некоторых указателей созданы дополнительные функции для удобства использования. К примеру, тип `char *` часто используется в C для хранения строк, и постоянно конвертировать вручную указатель на символ в строку и наоборот непрактично. Поэтому в Kotlin Runtime была добавлена возможность простого создания указателя на нативную строку по JVM-строке.

4. Особенности реализации

4.1. Загрузка деклараций из заголовков

Задача этого модуля — проиндексировать заголовочные файлы Objective-C-библиотеки и сериализовать все декларации в некоторый формат, десериализуемый далее в процессе построения иерархии и преобразования типов.

Разбор исходного кода на C или Objective-C — задача нетривиальная, но и не раз решённая. Поэтому для получения деклараций работа использует Clang [17], компилятор C/C++/Objective-C, а именно его публичный интерфейс в виде библиотеки `libclang` [18].

Поскольку `libclang` нативная библиотека, а компилятор Kotlin исполняется в JVM, появилась надобность в дополнительном слое, обеспечивающем взаимодействие компилятора и нативного кода. Этот слой реализован с помощью JNI (см. 2.1) и C++. Нативный метод в этом случае собирает с помощью `libclang` и сериализует всю необходимую в дальнейшем информацию.

В качестве формата сериализации был выбран двоичный формат Protocol Buffers [12]. Выбор обоснован удобством сериализации/десериализации (нет надобности реализовывать свой парсер) и скоростью работы. Был спроектирован `.proto`-файл, описывающий структуру данных для Protocol Buffers, по которому библиотека генерирует код нужных классов на языках C++ и Java.

Общий алгоритм

- На вход — пути к заголовочным файлам и дополнительные параметры командной строки, передаваемые компилятору Clang.
- Происходит вызов нативного метода из JVM. Передаваемые аргументы — пути и параметры, полученные на вход.
- Нативный код (компилируемый в итоге в библиотеку `KotlinNativeIndexer`) использует Higher level API [19] `libclang`

для индексирования деклараций. `libclang` производит лексический, синтаксический и семантический анализ заголовочных файлов. Полученные декларации `KotlinNativeIndexer` преобразует в сообщение[12] Protocol Buffers. При этом из Objective-C-деклараций индексируются:

  - Классы (`@interface`)
  - Протоколы (`@protocol`)
  - Категории (`@interface ... ( ... )`)
  - Методы в классах, протоколах и категориях и их параметры
  - Свойства (`@property`)
  - Перечисляемые типы (`enum`) и их константы

  Для каждой декларации сохраняется её имя, тип (если применимо), семантическое положение (к какому классу относится метод, к какому методу относится параметр) и другая информация.

- Полученное сообщение нативный код возвращает в виде массива байт в компилятор, исполняющийся на JVM. С помощью автоматически сгенерированных Java-классов сообщение десериализуется.

4.2. Kotlin Native

В целях обеспечения корректной работы отображённых Objective-C-классов в JVM, возникла необходимость в дополнительном уровне, обеспечивающем взаимодействие классов в JVM с их аналогами в Objective-C Runtime. Эта часть реализована с помощью JNI (см. 2.1), а нативный код на C++. Нативный код компилируется в библиотеку `libKotlinNative.dylib`, распространяемую вместе с Kotlin Runtime. Методы Kotlin Native вызываются из генерируемого кода (см. 4.3).

[12]Message в терминологии Protocol Buffers

Посылка сообщений

`objc_msgSend` — нативный метод, совершающий посылку Objective-C-сообщений. В качестве параметров он принимает получателя сообщения типа `ObjCObject`, имя метода в виде строки и массив аргументов, передаваемых методу, типа `java.lang.Object` и возвращает результат посылки сообщения, также типа `java.lang.Object`.

В нативном коде посылка сообщения осуществляется с помощью библиотеки libffi [20], предоставляющей кроссплатформенный интерфейс для вызова функций по указателю на функцию, информации о типах аргументов и возвращаемого значения и самим значениям аргументов. Для посылки сообщения происходит вызов функции `objc_msgSend` из Objective-C Runtime, принимающей те же аргументы, что и созданный JVM-аналог, но, разумеется, в нативном представлении.

Преобразование значения из `java.lang.Object` в нативный вид, имеющее место для значений аргументов вызываемой функции, для большинства типов тривиально: значения упакованных примитивных типов распаковываются (к примеру, у объекта типа `java.lang.Integer` вызывается метод `intValue()`), а у значений типов-указателей (`ObjCObject`, `Pointer` и т.д.) происходит доступ к полю, хранящему нативный указатель (у объекта типа `ObjCObject` доступ к полю `pointer`). Для значений функциональных типов (являющихся наследниками `Function`N) происходит создание нативной функции, делегирующей к методу `invoke` соответствующего функционального объекта (см. 4.2).

Обратное преобразование, из нативного типа в `java.lang.Object`, происходит для значения, которое вернула функция `objc_msgSend`: значения примитивных типов упаковываются, а по указателям создаётся экземпляр соответствующего класса, хранящий значение этого указателя. Отображение нативных функций (указатели на C-функции и блоки) в значения функциональных типов Kotlin в данной работе не поддерживается.

Создание функций

В Kotlin Native также была добавлена возможность создания нативной функции, которая с помощью JNI вызывает функциональный объект Kotlin, существующий в виде объекта класса `Function`N.

Реализации данного нативного метода требуется на этапе исполнения создать новую нативную функцию, т.е. выделить память, заполнить её необходимым кодом, пометить эту память как исполняемую и вернуть указатель на её начало. Для этого также используется библиотека libffi [20], а именно её часть Closure API. Она позволяет выполнить вышеописанное, при этом код создаваемой функции вызывает некоторый обработчик, передавая ему аргументы, указатель на результат, и любую пользовательскую информацию. Подобный обработчик в Kotlin Native обращается в JVM для вызова у объекта класса `Function`N метода `invoke`, что эквивалентно вызову функционального литерала в коде на Kotlin. Функциональный объект, его арность и типы аргументов передаются в пользовательской информации.

Значения переданных нативных аргументов конвертируются в соответствующие объекты JVM подобно тому, как это происходит с возвращаемыми значениями `objc_msgSend` (см. 4.2). И наоборот, результат выполнения функционального объекта конвертируется в нативное представление аналогично преобразованию аргументов при посылке Objective-C-сообщений.

Нативная функция не обязана выполняться в том же потоке в JVM, в котором была создана. JNI позволяет «прикрепить» нативный поток, созданный не из JVM, к виртуальной машине, в результате чего этому нативному потоку становится доступен весь интерфейс JVM. Данная функциональность также была реализована.

4.3. Генерация JVM-классов

Этот модуль компилятора генерирует байткод необходимых классов, полученных отображением из Objective-C.

В целях прозрачной работы приложения под JVM генерация байтко-

да соответствует конвенциям, принятым в компиляторе Kotlin: например, для каждого `trait`'а генерируется JVM-интерфейс, для каждого `class object` генерируется отдельный singleton-класс с финальным статическим полем `object$`, и т.д. Константы перечислимых типов генерируются в статические поля класса `objc.ObjcPackage` и по конвенции доступны как декларации верхнего уровня пакета `objc` в Kotlin.

Методы генерируемых классов делятся на абстрактные (в т.ч. в `trait`'ах) и конкретные (с реализацией). Каждый конкретный метод должен вести себя так же, как его нативный аналог в Objective-C. Для этого генерируется вызов нативного метода `objc_msgSend` из Kotlin Native (см. 4.2). Результирующий байткод каждого метода состоит из следующих этапов:

1. Загрузить аргументы для `objc_msgSend` на стек: получатель сообщения, имя сообщения, аргументы данного метода. Аргументы тривиально конвертируются к `java.lang.Object`: значения примитивных типов упаковываются в объекты соответствующих классов.

2. Вызвать `objc_msgSend` и получить результат выполнения на стеке. Нативный код обеспечит корректную конвертацию типов из JVM в нативное представление и обратно (см. 4.2).

3. Сконвертировать получившееся значение (типа `java.lang.Object`) к нужному типу и вернуть его из метода.

4.4. Работа с памятью

И в JVM, и в Objective-C Runtime существует механизм сборки мусора (garbage collection), осуществляющий удаление памяти, занимаемой объектами, на которые больше нет ссылок из программы. Поскольку с использованием данной работы появляется возможность наличия ссылок из одной системы на объекты в другой, необходимо сообщить сбор-

щику мусора последней о дополнительных ссылках, которые он иначе мог бы не учесть.

При передаче аргументов из JVM в Objective-C:

- Объекты примитивных типов, `ObjCObject` и `Pointer` конвертируются в соответствующие нативные типы на этапе вызова метода. Поэтому оригинальные JVM-объекты нативному коду не нужны, и можно ничего не предпринимать.

- Объекты функционального типа преобразуются в нативные функции, хранящие ссылку на функциональный объект, который необходимо вызвать. Однако ссылка на этот функциональный объект может остаться только в нативном коде. Для того, чтобы предотвратить преждевременное удаление объекта, в Kotlin Native используется механизм глобальных ссылок (global references) JNI [9]. Сборщику мусора в JVM не позволено удалять память, занимаемую объектом, до тех пор, пока на него есть хотя бы одна глобальная ссылка из нативного кода.

При передаче результата из Objective-C-метода в JVM, в случае если результат — Objective-C-объект, происходит вызов метода `- (id)retain`, сигнализирующего сборщику мусора Objective-C Runtime о том, что на объект есть ещё одна ссылка. В каждый из зеркалирующих классов в метод `finalize()` генерируются инструкции, вызывающие через Kotlin Native метод `- (void)release` на хранимом объекте. Таким образом, пока существует объект JVM-класса с указателем на объект Objective-C-класса, этот указатель корректен. Как только JVM-объект выходит из области видимости (на него не остаётся ссылок в JVM), счётчик ссылок объекта в Objective-C Runtime уменьшается на один, и, в случае достижения им нуля, сборщику мусора Objective-C Runtime позволяется очистить память, занимаемую этим объектом.

4.5. Тестирование

4.5.1. Функциональное тестирование

Было проведено обширное тестирование реализованного функционала. Каждый тест состоит из трёх файлов: заголовочный файл библиотеки с расширением `.h`, реализация библиотеки на Objective-C с расширением `.m` и клиентский код на Kotlin с расширением `.kt`.

Выполнение теста состоит из следующих частей:

1. Из заголовков и реализации внешним компилятором Objective-C строится динамическая библиотека.

2. Из заголовков и клиентского кода компилятором Kotlin строится клиентская программа.

3. Происходит запуск клиентской программы. Параметром окружения ей передаётся путь, в котором лежит скомпилированная на первом этапе библиотека.

Подобным образом были протестированы все отображения типов, создание/освобождение указателей, создание нативных функций и т.д.

Кроме этого, были реализованы небольшие примеры реального использования Objective-C-библиотек на Kotlin. Одна из тестовых программ успешно проигрывает звук из файла, путь к которому передан в командной строке, используя класс `NSSound` из библиотеки AppKit.

4.5.2. Сравнение производительности

Было проведено сравнение быстродействия данной работы и существующих аналогов (см. 2) на небольших искусственных примерах.

Реализованы следующие тесты:

1. **Counter** — тест на вызов метода без параметров и возвращаемого значения. Представляет из себя Objective-C-класс, хранящий числовую переменную, и экспортирующий два метода:

`- (void)increase` и `- (int)get`. Тестирующий код в цикле вызывает `increase` у экземпляра этого класса (и дополнительно `get` для проверки правильности полученного значения).

2. **Sum** — тест на скорость конвертации значений примитивных типов. Тестируемый код в цикле вызывает метод с пятью аргументами типов `int`, `short`, `long`, `float` и `double`, и получает результатом типа `double` сумму пяти чисел.

3. **Singleton** — тест на скорость конвертации указателей на Objective-C-объекты. В тесте в цикле вызывается метод `+ (Singleton *)getInstance` тестового класса, возвращающий созданный заранее экземпляр класса.

4. **Mirror** — тестовый метод `+ (id)get: (id)object` принимает параметром указатель на любой Objective-C-объект и возвращает его же.

5. **Callback** — тест на быстродействие динамически конструируемых функций. Тестовый метод `+ (void)invoke: (void (*)())callback` принимает указатель на функцию без параметров и возвращаемого значения и вызывает эту функцию.

Тестирование проводилось на MacBook Air с процессором 1.8 GHz Intel Core i7, оперативной памятью 4 GB 1333 MHz DDR3, на операционной системе Mac OS X 10.7.5. Временем работы каждого испытания считается среднее время работы по нескольким запускам. В рамках одного запуска тестирующий код в цикле вызывает метод тестового класса. Количество итераций цикла в тестах Counter и Sum равно $5 \cdot 10^5$, в тестах Singleton, Mirror и Callback — 10^5.

Результаты тестирования приведены в таблице 4. В тестировании помимо данной работы (обозначенной в таблице как Kotlin) принимали участие библиотеки JNA, BridJ и Rococoa.

Выводы Несмотря на то, что в работе не ставилась целью высокая производительность, итоговая система позволяет использовать

	JNA	**BridJ**	**Rococoa**	**Kotlin**
Counter	2.93	0.53	2.19	0.62
Sum	4.06	0.43	4.50	4.59
Singleton	0.99	0.37	2.79	0.53
Mirror	1.06	0.49	2.67	0.75
Callback	1.26	0.41	1.69	1.99

Таблица 4: Результаты тестирования производительности (время в секундах)

Objective-C-библиотеки из JVM примерно с такими же временными затратами, как и другие аналогичные продукты.

Кроме того, следует отметить простоту получающегося клиентского кода. Например, тестовая оболочка, реализованная для проведения вышеописанных испытаний, для библиотеки Rococoa занимает семь файлов суммарным объёмом 141 строка, а для BridJ — шесть файлов и 142 строки. Значительная часть этого кода содержит лишь перечисление нативных деклараций для корректной интерпретации библиотекой. Аналогичный тестирующий код для данной работы лишён этого недостатка и занимает один файл в 58 строк.

Заключение

В рамках данной работы решалась задача прозрачного взаимодействия кода на Kotlin с динамическими библиотеками на Objective-C. Были изучены особенности обоих языков и продумано отображение понятий одного языка в понятия другого.

Работа реализована в виде нескольких модулей компилятора Kotlin в байткод JVM.

- Реализован модуль, загружающий декларации из заголовков библиотеки.
- Реализован модуль, отображающий декларации в термины Kotlin.
- Реализован модуль, обеспечивающий генерацию корректных JVM-классов.

Реализованная функциональность протестирована на соответствие минимальным функциональным требованиям и на производительность. По результатам сравнения с существующими аналогами получена сопоставимая производительность и существенный выигрыш в размере необходимого клиентского кода.

Список литературы

[1] BridJ. — 2013. — URL: https://code.google.com/p/bridj/ (online; accessed: 30.05.2013).

[2] Foundation Framework Reference. — 2013. — URL: https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/ObjC_classic/Intro/IntroFoundation.html (online; accessed: 30.05.2013).

[3] Growl. — 2013. — URL: http://growl.info/ (online; accessed: 30.05.2013).

[4] JNA API. — 2013. — URL: http://twall.github.io/jna/3.5.2/javadoc/ (online; accessed: 30.05.2013).

[5] JNAerator. — 2013. — URL: https://code.google.com/p/jnaerator/ (online; accessed: 30.05.2013).

[6] Java Native Access (JNA). — 2013. — URL: https://github.com/twall/jna (online; accessed: 30.05.2013).

[7] Java SE Documentation: Java Native Interface. — 2013. — URL: http://docs.oracle.com/javase/6/docs/technotes/guides/jni/ (online; accessed: 30.05.2013).

[8] Kotlin. — 2013. — URL: http://kotlin.jetbrains.org/ (online; accessed: 30.05.2013).

[9] Liang Sheng. The Java™ Native Interface: Programmer's Guide and Specification. — Addison-Wesley Professional, 1999. — ISBN: 0201325772.

[10] Matt Gallagher. What is a meta-class in Objective-C? // Cocoa with Love. — 2010. — January. — URL: http://www.cocoawithlove.com/2010/01/what-is-meta-class-in-objective-c.html.

[11] Objective-C Runtime Programming Guide: Type Encodings. — 2013. — URL: http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html (online; accessed: 30.05.2013).

[12] Protocol Buffers: Google's data interchange format. — 2013. — URL: https://code.google.com/p/protobuf/ (online; accessed: 30.05.2013).

[13] Rococoa — A generic Java wrapper for Cocoa on Mac OS X. — 2013. — URL: https://code.google.com/p/rococoa/ (online; accessed: 30.05.2013).

[14] Swing. — 2013. — URL: http://docs.oracle.com/javase/6/docs/technotes/guides/swing/ (online; accessed: 30.05.2013).

[15] System V Application Binary Interface. — 2007. — URL: http://www.cs.tufts.edu/comp/40/readings/amd64-abi.pdf (online; accessed: 30.05.2013).

[16] Xcode. — 2013. — URL: https://developer.apple.com/xcode/ (online; accessed: 30.05.2013).

[17] clang: a C language family frontend for LLVM. — 2013. — URL: http://clang.llvm.org/ (online; accessed: 30.05.2013).

[18] libclang: C Interface to Clang. — 2013. — URL: http://clang.llvm.org/doxygen/group__CINDEX.html (online; accessed: 30.05.2013).

[19] libclang: Higher level API functions. — 2013. — URL: http://clang.llvm.org/doxygen/group__CINDEX__HIGH.html (online; accessed: 30.05.2013).

[20] libffi: A Portable Foreign Function Interface Library. — 2013. — URL: http://sourceware.org/libffi/ (online; accessed: 30.05.2013).