UNIVERSITÀ
DI PARMA

Dipartimento di Scienze Matematiche,
Fisiche e Informatiche

Corso di Laurea in Informatica

Tesi di Laurea

Applicazioni iOS: analisi dei linguaggi di programmazione utilizzati e sviluppi futuri della piattaforma

Candidato:
Federico Bertoli

Relatore:

Prof. Roberto Alfieri

Anno Accademico 2015/2016

Indice

TODO

Introduzione

Lo sviluppo di applicazioni in ambito mobile ha subito una crescita esponenziale da quando, nel 2008, Apple ha permesso di accedere agli strumenti di sviluppo interni (nello specifico il software Xcode e i relativi SDK) e ha inserito nella versione 2.0 del software iOS l'applicazione App Store, permettendo di fatto a chiunque di rendere la propria idea una realtà, supportata da un'infrastruttura di servizi ben collaudata, concentrando i propri sforzi solamente sullo sviluppo del software.

In concomitanza anche Google,tramite l'Android Market, ha permesso la pubblicazione di applicazioni sul proprio store, creando così una "corsa all'oro" che solamente ora, 8 anni dopo, sta subendo un sensibile calo dopo anni di crescita significativa e costante.

Precedentemente a queste due piattaforme lo sviluppo in ambito mobile era frenato da fattori importanti quali la mancanza di software di sviluppo stabili e continuamente aggiornati, i sistemi operativi non sufficientemente avanzati e la scarsa potenza dei dispositivi dell'epoca.

Ciò che ha portato a questa tesi è stato un interesse molto forte verso questo mondo con nemmeno una decade sulle spalle (se si parla di smartphone) e ancora pieno di sviluppi, la maggior parte di questi solo annunciati e ancora in piena fase di progettazione (si pensi per esempio alla realtà aumentata, alle funzionalità appena annunciate sui nuovi dispositivi come le doppie fotocamere, i microfoni HAAC o i sensori per il touchscreen in 3 dimensioni).

In questa tesi abbiamo focalizzato l'attenzione sull'ecosistema creato da Apple per i propri dispositivi iOS ed in particolare sui linguaggi utilizzati per lo sviluppo, Objective-C e Swift.

Capitolo 1

Objective-C

Creato agli inizi degli anni '80, Objective-C ha guadagnato la sua popolarità come il linguaggio principale per il sistema operativo NEXTSTEP.
Quando nel 1996 Next venne acquisita da Apple il loro sistema operativo divenne la base del nuovo Mac OS, rendendo così questo linguaggio la chiave per lo sviluppo negli anni a venire. Per molti anni Objective-C è rimasto l'unico linguaggio per lo sviluppo su piattaforme Mac ed iOS.
La sua implementazione è basata sul linguaggio C con l'aggiunta di caratteristiche di programmazione orientata agli oggetti ed un runtime dinamico.

1.1 Caratteristiche del linguaggio

Questo capitolo descrive le funzionalità che il linguaggio aggiunge allo standard C.

1.1.1 Sintassi

Implementazione di una classe, i file .h e .m

L'implementazione di una classe avviene attraverso due file distinti:

- Un file con estensione .h, che conterrà la dichiarazione dell'interfaccia di classe, i metodi e le properties pubbliche.
- Un file con estensione .m, che conterrà l'implementazione della classe, la definizione di metodi e properties private; in questo file inoltre saranno definiti i metodi e le variabili d'istanza private.

Esempio di file header (.h) di una classe:

```
#import <Foundation/Foundation.h>

@interface Persona: NSObject ()

//costruttore personalizzato con parametri

- (id)initConNome:(NSString *)unNome
      cognome:(NSString *)unCognome;

// dichiarazione delle properties private

@private
        NSString *_nome;
        NSString *_cognome;
        int _eta;
}

// dichiarazione dei metodi getter e setter

- (NSString *)nome;
- (void)setNome:(NSString *)nome;
- (NSString *)cognome;
- (void)setCognome:(NSString *)cognome;
- (int)eta;
- (void)setEta:(int)eta;

@end
```

Esempio di file di implementazione (.m) di una classe:

```
#import "Persona.h"

@implementation Persona

//implementazione della classe dichiarata in Persona.h

- (id)initConNome:(NSString *)unNome
      cognome:(NSString *)unCognome {

    if( self = [super init] )
    {
        _nome = unNome;
        _cognome = unCognome;
    }
```

```
    return self;
}

-(NSString *) nome {
        return _nome
}

-(NSString *) cognome {
        return _cognome
}

-(int) eta {
        return _eta
}

-(void)setNome:(NSString *)nome {
        _nome = nome;
}

-(void)setCognome:(NSString *)cognome {
        _cognome = cognome;
}

-(void)setEta:(int)eta {
        _eta = eta;
}
@end
```

Attraverso la parola chiave @property è anche possibile dare direttiva al compilatore di definire i setter e i getter:

```
//come dichiarati in Persona.h
- (NSString *)nome;
- (void)setNome:(NSString *)nome;

//con property diventa
@property NSString * nome;
```

Inoltre, attraverso la parola chiave @synthesize, è possibile indicare al compilatore di implementare i setter e i getter di una property che sia stata definita precedentemente:

```
//come dichiarati in Persona.m

-(NSString *) nome {
        return _nome
}

-(void)setNome:(NSString *)nome {
        _nome = nome;
}
//con synthesize diventa:

@synthesize nome = _nome;
```

Le properties permettono di accedere ai getter e ai setter utilizzando la notazione puntata, per una maggiore leggibilità del codice:

```
//senza properties:

int eta = [persona eta];
[persona setEta:22];

//con properties:

int eta = persona.eta;
persona.eta = 22;
```

Dichiarazione e definizione dei metodi

Come visto per la definizione dei setter, una dichiarazione di funzione (metodo) ha la seguente sintassi:

```
//Nell'ordine: (tipoDiRitorno) nomeMetodo:(tipoArg1)nomeArg1;

- (NSInteger)calcolaEta:(NSDate)dataDiNascita;
```

Definizione del metodo appena dichiarato:

```
- (NSInteger)calcolaEta:(NSDate)dataDiNascita {

        NSDate* oggi = [NSDate date];
        NSDateComponents* componentiCalendario = [[NSCalendar currentCalendar]
                                                  components:NSCalendarUnitYear
                                                  fromDate:dataDiNascita
                                                  toDate:oggi
                                                  options :0];

        NSInteger eta = [componentiCalendario year];

        return eta;
}
```

Invio dei messaggi

In Objective-C, a differenza della maggior parte degli altri linguaggi, un metodo non viene chiamato direttamente, ma si invia un messaggio all'oggetto stesso:

```
[persona calcolaEta:dataDiNascita];
```

Questo è reso possibile dal runtime del linguaggio che, tramite una lista, tiene traccia di tutti i metodi e funzioni che conosce. Ogni elemento della lista è composto da due campi: il nome del metodo (conosciuto come il selector dello stesso) e la sua locazione in memoria.
Durante la fase di compilazione del codice sorgente, quando vi è la chiamata di un metodo, il compilatore sostituisce il frammento di codice nel seguente modo:

```
[persona calcolaEta:dataDiNascita];
```

in

```
objc_msgSend(persona, @selector(calcolaEta:),dataDiNascita);
```

La funzione objc_msgSend() opera tramite un lookup dinamico: conoscendo il nome del metodo da ricercare scorre la lista per verificare la sua effettiva presenza e, in caso affermativo, lo esegue.
Questo comportamento ha caratteristiche particolari: potremmo far puntare un certo selettore A, che prima puntava al codice per A, ad un codice per un certo B.
Lo svantaggio è che il tempo di esecuzione è leggermente maggiore di una chiamata diretta alla parte di codice; si tratta comunque di nanosecondi di differenza.

Inoltre il runtime, prima di inviare il messaggio, richiede all'oggetto se il metodo è riconosciuto dallo stesso. Questo significa che l'oggetto può decidere se accettare il messaggio (è anche per questo che si possono inviare messaggi a nil), inoltrarlo ad un oggetto differente, decidere di eseguire codice differente per un metodo specifico.
Più in dettaglio, quando inviamo un messaggio ad un oggetto, non abbiamo garanzie sul fatto che:

-Il metodo che andiamo a chiamare non sia necessariamente quello che verrà effettivamente chiamato

-L'oggetto che riceverà il messaggio non sarà necessariamente quello che vogliamo che risponda.

1.1.2 Compilatore

L'IDE di Apple, Xcode, utilizza clang. Quest'ultimo è un compilatore front end per C, C++, Objective-C, Objective-C++, OpenMP, OpenCL e CUDA, basato sul compilatore LLVM (Low Level Virtual Machine).
LLVM in origine doveva utilizzare il front end di GCC, ma dopo aver riscontrato problemi di integrazione e di sviluppo l'azienda ha deciso di crearne iternamente uno proprio, clang appunto, reso open-source nel Giugno 2007.

1.1.3 Utilizzo in iOS e frameworks Cocoa

Il linguaggio principale di tutti i frameworks (quali CoreOS, Foundation, CocoaTouch) di iOS e MacOS è ancora Objective-C come dal principio, in quanto Swift non ha ancora raggiunto un livello di maturità tale per essere impiegato per questi scopi (la versione 3 del linguaggio non ha ancora ABI stabili e la sintassi è ancora in corso di modifiche).L'intenzione di Apple è

quella di mantenere e migliorare i due linguaggi parallelamente per ancora molto tempo.

Capitolo 2

Swift

Swift è stato introdotto nel Giugno 2014 durante l'annuale conferenza per gli sviluppatori di Apple, e reso open source nel Dicembre 2015.
Questo linguaggio è una combinazione delle migliori caratteristiche di C ed Objective-C in aggiunta a features che lo rendono un linguaggio moderno e sicuro, rendendo lo sviluppo più rapido ed efficiente.
La sintassi semplificata rispetto ad Objective-C ed il completo supporto alle API di Cocoa e Cocoa Touch lo rendono fruibile anche agli sviluppatori alla prime armi con le piattaforme di Apple.

2.1 Caratteristiche

2.1.1 Sintassi

Implementazione di una classe

In Swift, al contrario di Objective-C, non esistono due file distinti per l'interfaccia e l'implementazione, ma uno solo con l'estensione .swift
Esempio di creazione di una nuova classe:

```
class Persona {

//dichiarazione delle properties

        var nome: String?
        var cognome: String?
        var eta: Int?

//costruttore personalizzato con parametri
```

```
        init (nome: String, cognome: String, eta: Int) {
                self.nome = nome
                self.cognome = cognome
                self.eta = eta
        }

//dichiarazione dei metodi setter e getter

        func getNome() -> String {

                return self.nome
        }

        func setNome(nome: String) {

                self.nome = nome
        }

        func getCognome() -> String {

                return self.cognome
        }

        func setCognome(cognome: String) {

                self.cognome = cognome
        }

        func getEta() -> Int {

                return eta
        }

        func setEta(eta: Int) {

                self.eta = eta
        }
}
```

Dichiarazione e definizione dei metodi

Una dichiarazione di funzione (metodo) ha la seguente sintassi:

```
//Nell'ordine: func nomeMetodo(nomeArg1:tipoArg1) -> tipoDiRitorno

func calcolaEta(dataDiNascita: NSDate) -> Int
```

Definizione del metodo appena dichiarato:

```
func calcolaEta(dataDiNascita: NSDate) -> Int

        let oggi = Date()

    let componentiCalendario = Calendar.current.dateComponents([.year], from:
            dataDiNascita, to: oggi)

        let eta = componentiCalendario.year!

        return eta;
}
```

Le funzioni in Swift sono trattate come oggetti, ciò significa che una funzione può ritornare un'altra funzione:

```
func creaIncrementatore() -> ((Int) -> Int) {

        func aggiungiUno(numero: Int) -> Int {

                return 1 + numero

        }

        return aggiungiUno
}

var incrementatore = creaIncrementatore()
incrementatore(7)
```

Closures

Le funzioni sono un caso speciale di "*closure*": il codice in una *closure* ha accesso a variabili e funzioni che sono disponibili nel suo scope, anche se viene eseguita in uno scope diverso. Una *closure* viene definita dalla sintassi { }, utilizzando il separatore *in* per gli argomenti e il tipo di ritorno dal corpo:

```
var numeri = [2,25,21,89,90]

numeri.map({
        (numero: Int) -> Int in
        let risultato = 3 * numero
        return risultato
})
```

Esistono varie modalità per scrivere le *closure*: quando il tipo è già conosciuto, come per esempio in una *callback* per un *delegate*, si possono omettere i tipi dei parametri e il tipo di ritorno (o entrambi) nel caso in cui ci sia un singolo statement, in quanto la *closure* ritorna implicitamente il valore di ritorno:

```
let numeriInMap = numeri.map({ numero in 3 * numero })
```

Ci si può riferire ai parametri per numero invece che per nome, approccio utile specialmente in closure che richiedono poco codice; una closure passata come ultimo argomento di una funzione può essere scritta immediatamente dopo le parentesi e, se quest'ultima è l'unico argomento della funzione stessa, si possono omettere le parentesi tonde:

```
//ordino i numeri in modo crescente

let numeriOrdinati = numeri.sorted { $0 > $1 }
```

Enumerazioni

La sintassi enum è utilizzata per dichiarare le enumerazioni in Swift. La particolarità rispetto ad Objective-C è che queste possono contenere metodi:

```
enum TipologieDiCase {

        case condominio, villa, indipendente, attico

        func descrizione() -> String {

                switch self {

                case .condominio:
                        return "Condominio"
                case . villa :
                        return "Villa"
                case .indipendente:
                        return "Casa indipendente"
                case .attico
                        return "Attico"
                default:
                        return String(self.rawValue)

                }
        }
}

let villa = TipologieDiCase.villa
```

```
let descrizioneVilla = villa.descrizione()
```

Protocolli ed estensioni

Un protocollo definisce un'interfaccia di metodi, variabili ed altri eventuali requisiti che definiscono una particolare funzionalità. Quest'ultimo può essere quindi adottato da una classe, struct o enum che ne forniranno l'implementazione:

```
protocol ProtocolloDiNavigazioni {

        func navigaAlleImpostazioni(sender: CollectionView)
        func navigaAlDettaglioEvento(sender: UICollectionView)
}

class MenuPrincipale: UICollectionViewController, UICollectionViewDelegateFlowLayout:
        ProtocolloDiNavigazioni {

        override func viewDidLoad() {
        super.viewDidLoad()
                collectionView?.delegate = self
                collectionView?.dataSource = self
        }

        ...

        //MARK: CollectionViewDelegate

        override func collectionView(collectionView: UICollectionView,
                didSelectItemAtIndexPath indexPath: NSIndexPath) {

                switch indexPath {

                        case 0:
                                navigaAlleImpostazioni(self.collectionView)
                        case 1:
                                navigaAlDettaglioEvento(self.collectionView)
                }
        }

        //MARK: Implementazione del protocollo

        func navigaAlleImpostazioni(sender: UICollectionView) {

        appDelegate.gotoSettingsVC()
    }

    func navigaAlDettaglioEvento(sender: UICollectionView) {
```

```
        appDelegate.gotoEventDetailVC()
    }
}
```

Le estensioni sono invece un modo per aggiungere funzionalità ad un tipo esistente, come nuovi metodi e *computed properties*:

```
//Estensione che aggiunge un effetto di blur ad una imageView

extension UIImageView
{
    func aggiungiBlur()
    {
        let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.Light)
        let blurEffectView = UIVisualEffectView(effect: blurEffect)
        blurEffectView.frame = self.bounds
        self.addSubview(blurEffectView)
    }
}

let containerImmagine = UIImageView()
containerImmagine.image = UIImage(named: "beer2beerlogo.jpg")
containerImmagine.aggiungiBlur()
```

Gestione degli errori

Gli errori vengono rappresentati utilizzando un qualsiasi tipo che si conformi al protocollo Error; la parola chiave throws viene utilizzata per indicare che una funzione può ritornare un errore, utilizzando la parola chiave throw. Se un errore viene lanciato dall'interno di una funzione, quest'ultima ritorna immediatamente e l'errore viene gestito dalla funzione chiamante:

```
enum ErroriStampante: Error {

        case cartaEsaurita
        case inchiostroEsaurito
        case cassettoChiuso

}

func invia(lavoro: Int, allaStampante nomeStampante: Stringa) throws -> String {
        if nomeStampante = "Rusty old printer" {

                throw ErroriStampante.cassettoChiuso
        }
```

```
        return "Lavoro inviato alla stampante"
}

        do {

        let rispostaStampante = try invia(lavoro: 2303, allaStampante: "Sala meeting")

        print(rispostaStampante)

        } catch {

                print(error)
        }
```

Si possono inoltre utilizzare più blocchi catch per gestire errori specifici:

```
        do {

                let rispostaStampante = try invia(lavoro: 2303, allaStampante: "Sala
                        meeting")

                print(rispostaStampante)

                } catch ErroriStampante.cassettoChiuso {

                        print("Aprire il cassetto")
                }
```

Un altra modalità per la gestione degli errori è l'utilizzo della la parola chiave try? per convertire il risultato in un tipo optional: se la funzione lancia un errore, questo specifico errore è ignorato e la funzione ritorna nil; alternativamente il risultato è un optional contenente il valore ritornato dalla funzione:

```
let foglioStampato = try? invia(lavoro: 1984, allaStampante: "Sala meeting")
let erroreDiStampa = try? invia(lavoro: 1948, allaStampante: "Rusty old printer")
```

Generics

Una funzione generica, più comunemente chiamata template, è così dichiarata:

```
func creaArray<Elemento>(ripeti elemento: Elemento, numeroDiVolte: Int) -> [Elemento] {

        var risultato = [Elemento]()

        for _ in 0..< numeroDiVolte {
```

```
                risultato.append(elemento)
        }

        return risultato
}

//chiamata alla funzione

creaArray(ripeti: "Tick tock", numeroDiVolte: 3)
```

Si possono implementare generics di funzioni o di classi, enumerazioni e struct:

```
//Reimplementazione del tipo optional della libreria standard di Swift

enum OptionalValue<Wrapped> {

        case none
        case some(Wrapped)
}

var possibileIntero: OptionalValue<Int> = .none
possibileIntero = .some(100)
```

Si utilizza la parola chiave where prima del corpo per indicare una lista di requisiti, per esempio per indicare che il tipo deve conformarsi ad un protocol, per richiedere che due tipi siano uguali o per indicare che una classe deve avere una particolare superclasse:

```
func elementiComuni<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool {

        where T.Iterator.Element: Equatable,
                T.Iterator.Element == U.Iterator.Element {

                for lhsItem in lhs {

                        for rhsItem in rhs {

                                if lhsItem == rhsItem {

                                        return true
                                }
                        }
                }
        }

        return false
}
```

```
//chiamata alla funzione

elementiComuni([1,2,3],  [3])
```

Tuples

Feature non presente in Objective-C, le tuple raggruppano più valori (di un tipo qualsiasi o tipi differenti) in un singolo valore composto. Per esempio, potremmo descrivere lo status code 404 dell'HTTP con una tupla in questo modo:

```
let  errore404http = (404, "Not found")
```

La tupla (404, "Not found") raggruppa insieme un tipo Int e uno String; si può creare qualsiasi permutazione di tipi. Per ottenere i singoli valori da una tupla, quest'ultima deve essere scomposta:

```
let  (statusCode, statusMessage) = errore404http
```

Questo particolare tipo è utile come valore di ritorno dalle funzioni, per esempio una funzione che ha il compito di caricare una pagina web potrebbe usare la tupla sopra descritta per indicare il successo o il fallimento del caricamento, fornendo più informazioni rispetto ad un valore di ritorno singolo di un singolo tipo.

Optionals

Questo tipo è un pilastro portante di Swift, in quanto viene utilizzato in tutti i casi in cui il valore di ritorno potrebbe essere nullo (nil). La logica di funzionamento è la seguente: o esiste un tipo di ritorno, e quindi si utilizza l'unwrapping per accedere al valore, o non c'è valore alcuno.
Il concetto di optional non esiste in C o Objective-C. Ciò che ci si avvicina maggiormente è l'abilità di ritornare nil da un metodo che altrimenti ritornerebbe un oggetto, con nil a significare l'assenza di un oggetto valido.
Quanto appena descritto vale solamente per gli oggetti (non structs, tipi C o enumerativi); per questi i metodi Objective-C ritornano solamente un valore speciale (come per esempio NSNotFound). Ciò implica che il chiamante dei metodi sappia che c'è uno speciale valore da verificare, mentre l'approccio di Swift permette di indicare l'assenza di qualsiasi valore in assoluto, senza la necessità per speciali costanti.
In questo esempio vediamo come gli optional possano essere utilizzati per gestire il caso di assenza di valore. Il tipo Int di Swift ha un costruttore

che converte un tipo String in un valore di tipo Int; questa conversione può fallire, perciò viene utilizzato un tipo optional:

```
let possibileNumero = "123"
let numeroConvertito = Int(possibileNumero)
//numeroConvertito e' di tipo Int?, che si legge come "optional Int"
```

Poichè il costruttore può fallire, questi ritorna un tipo optional Int, indicato da Int?. Il punto di domanda indica che il valore contiene un tipo optional, a significare che potrebbe contenere un valore Int o nessun valore.
Utilizzando la speciale parola chiave nil, si indica che un tipo optional non ha valore alcuno. nil non può essere utilizzato con constanti e variabili non optional.

```
var codiceRispostaServer: Int? = nil
```

Se viene creata una variabile optional, alla quale non si assegna alcun valore, a quest'ultima viene automaticamente assegnato nil.
Il nil di Swift non è però equivalente a quello di Objective-C: in quest ultimo, nil è un puntatore ad un oggetto non esistente, in Swift non è invece un puntatore, ma l'assenza di valore alcuno. Optionals di qualunque tipo possono essere settati a nil, non solamente oggetti.
Per verificare la presenza di valore in un tipo optional è possibile usare lo statement if in questo modo:

```
if numeroConvertito != nil {

        //numero convertito contiene un valore
}
```

Alternativamente è possibile utilizzare il simbolo !, indicato come *forced unwrapping* del tipo optional; questo approccio è rischioso e porta spesso ad errori in runtime. Viene utilizzata molto più frequentemente la sintassi dell'optional binding, per verificare se una certa variabile optional contiene un valore e, se presente, assegnarlo ad una costante o variabile temporanea. Viene utilizzato con gli statement if e while:

```
if let numero = Int(possibileNumero) {
        //se entro nel blocco significa che possibileNumero
        //contiene un intero poiche' la conversione ad Int va a buon fine
} else {
        //possibileNumero non e' stato convertito ad Int
}
```

Il codice può essere interpretato in questo modo: se l'optional Int ritornato da Int(possibileNumero) contiene un valore, allora assegna alla nuova costante

chiamata numero il valore contenuto nell'optional. Questa costante è già stata inizializzata con il valore contenuto all'interno dell'optional, quindi non è necessario utilizzare la sintassi ! per forzare l'accesso al valore. In uno statement si possono concatenare più optional binding separati da virgola e, se almeno uno di questi valori è nil o una condizione booleana valuta a false, l'intera condizione dello statement viene considerata falsa.
Le costanti e le variabili create tramite l'optional binding in uno statement if sono accessibili solamente all'interno dello statement stesso; per permettere l'accesso anche alle linee di codice che seguono lo statement, è necessario utilizzare lo statement guard.

2.1.2 Compilatore e libreria standard

Architettura del compilatore

Il compilatore di Swift è composto dai seguenti componenti principali:

- Parser: è il componente responsabile della generazione dell'albero di sintassi astratta senza alcuna informazione semantica o di tipo, e genera errori in caso di problemi grammaticali nel sorgente.

- Analizzatore semantico: trasforma l'albero generato dal parser in un albero ben formato e con controllo sui tipi. Questa analisi include la type inference e, in caso di successo, indica che è sicuro generare il codice dall'albero appena creato.

- Clang importer: importa i moduli di Clang e mappa le API C o Objective-C nelle rispettive API Swift. Gli alberi risultanti vengono utilizzati dall'analizzatore semantico.

- Generatore SIL: SIL è l'acronimo di Swift Intermediate Language, ovvero un linguaggio intermedio di alto livello, specifico per l'analisi e l'ottimizzazione del codice. Questa fase trasforma l'albero di sintassi astratta creato dall'analizzatore in un cosiddetto SIL "grezzo".

- Trasformazioni SIL: questo strumento esegue ulteriori diagnostiche che influenzano la correttezza del programma (come ad esempio l'uso di variabili non inizializzate). Il risultato finale di queste trasformazioni è SIL "canonico".

- Ottimizzazioni SIL: questa fase esegue ulteriori ottimizzazioni specifiche di alto livello, quali Automatic Reference Counting per la gestione della memoria, devirtualizzazione e specializzazione dei tipi generics.

- Generazione IR LLVM: IR significa Intermediate Representation, o rappresentazione intermedia. Questa fase trasforma il SIL in LLVM IR e, giunti a questa fase, LLVM procede ad ottimizzare il codice e genera il codice macchina.

Libreria standard Swift

La libreria standard Swift comprende un certo numero di tipi di dati, protocolli e funzioni, compresi i tipi fondamentali (ad esempio, Int, Double), collezioni (Array, Dizionario) insieme ai protocolli che li descrivono e gli algoritmi che operano su di essi, i caratteri, le stringhe e le primitive di basso livello (ad esempio, UnsafeMutablePointer).
Il repository della libreria standard viene ulteriormente suddiviso:

- Nucleo principale: include la definizione di tutti i tipi, protocolli e funzioni.
- Runtime: Il runtime di supporto è il componente che risiede nel mezzo tra il compilatore ed il nucleo della libreria standard. E' il responsabile dell'implementazione delle features dinamiche del linguaggio, come il casting (ad esempio per gli operatori as! ed as?), i metadata dei tipi (per supportare i generics e la reflection) e la gestione della memoria (allocazione degli oggetti, reference counting). Differentemente dalle altre librerie di alto livello, il runtime è scritto quasi esclusivamente in linguaggio C++ o Objective-C.
- Overlays per SDK: componenti specifici per le piattaforme Apple, forniscono modifiche ed aggiunte specifiche per Swift ai framework Objective-C, per migliorarne la loro interoperabilità.

Whole module optimization

Con la versione 3.0 di Swift è stata introdotta una modalità di ottimizzazione del compilatore che, dipendentemente dal progetto, permette miglioramenti delle performance significativi.
Senza WMO viene effettuata una compilazione a singolo modulo: questi è un set di files Swift, ed ognuno viene compilato in una singola unità di distribuzione (un framework o un eseguibile). Nella compilazione a file singolo il compilatore è invocato separatamente per ogni file nel modulo: dopo la lettura ed il parsing del singolo file, il compilatore ottimizza il codice, genera il codice macchina e scrive il file oggetto corrispondente. Successivamente il linker unisce i file oggetto e genera la libreria condivisa o l'eseguibile.

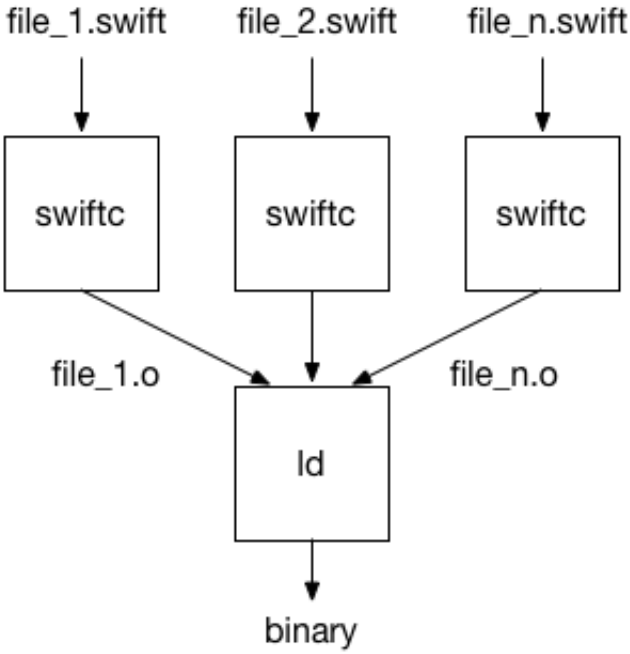

Figura 2.1: *Visualizzazione del lavoro del compilatore in modalità di compilazione per ogni singolo file*

Questa compilazione singola limita le ottimizzazioni inter funzione (come l'inlining o la specializzazione dei generics) alle funzioni chiamate all'interno dello stesso file. Come esempio, assumiamo che un file di un modulo (chiamato utils.swift), contenga una struttura dati generica (chiamata Container<T>), con un metodo chiamato getElement. Questo metodo è chiamato nel modulo, ad esempio nel file main.swift:

```
//main.swift:

func add (c1: Container<Int>, c2: Container<Int>) -> Int {
  return c1.getElement() + c2.getElement()
}
```

```
// utils . swift :

struct Container<T> {
  var element: T

  func getElement() -> T {
```

```
    return element
  }
}
```

Quando il compilatore ottimizza il file main.swift non conosce come la funzione getElement sia implementata, conosce solo il fatto che è presente; viene quindi generata una chiamata al suddetto metodo.
Analogamente, quando il compilatore ottimizza il file utils.swift non conosce per quale tipo concreto il metodo viene chiamato, quindi può generare solamente una versione generica della funzione, rendendo il codice più lento rispetto ad uno ottimizzato per un tipo concreto.
Anche solamente uno statement return in getElement necessita di un lookup di tipo per verificare come copiare l'elemento; può trattarsi di un tipo Int o di un tipo più dispendioso in termini di risorse che richiede operazioni di reference counting.
Con l'ottimizzazione per l'intero modulo (WMO), invece, il compilatore ottimizza tutti i file di un modulo nella loro interezza:

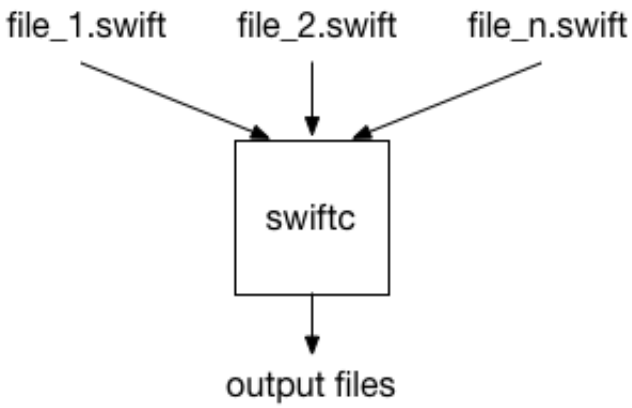

Figura 2.2: *Visualizzazione del lavoro del compilatore in modalità di compilazione whole module optimization*

Questo porta due notevoli vantaggi:
Il compilatore vede le implementazioni di tutte le funzioni del modulo, e può quindi procedere con le ottimizzazioni specifiche; specializzare una funzione significa che il compilatore crea una nuova versione del metodo specifica per il contesto della chiamata. Ad esempio, il compilatore può ottimizzare le funzioni di tipo generics per i tipi concreti.

Nell'esempio, il compilatore crea una versione della struct Container specifica per i tipi Int:

```
struct Container {
  var element: Int

  func getElement() -> Int {
    return element
  }
}
```

Può quindi procedere all'inlining del metodo specializzato getElement all'interno della funzione add:

```
func add (c1: Container<Int>, c2: Container<Int>) -> Int {
  return c1.element + c2.element
}
```

Contrariamente alla compilazione a file singolo, il lavoro viene eseguito in poche istruzioni macchina. La specializzazione di funzioni e l'inlining tra più file sono solo esempi delle ottimizzazioni che il compilatore può apportare con l'approccio WMO; un altro campo in cui può migliorare le performance è nella gestione della memoria, in quanto il compilatore può rimuovere operazioni di reference counting ridondanti.
Un altro beneficio è che il compilatore può valutare tutti gli usi di funzioni non pubbliche e, con questa informazione, eliminare le funzioni che non vengono mai utilizzate, rese in questo stato come side-effect derivante dalle altre ottimizzazioni. Riprendendo il codice di esempio, possiamo assumere che la funzione add sia l'unica ad utilizzare Container.getElement. Dopo aver effettuato l'inlining della funzione getElement, questa non viene più utilizzata e può quindi essere rimossa. Anche se il compilatore dovesse decidere di non procedere con l'inlining della funzione, potrebbe comunque rimuovere la versione generica di getElement, poichè la funzione add utlizzerebbe solamente la versione specializzata.

Tempo di compilazione

Con l'ottimizzazione a file singolo il driver del compilatore inizia la compilazione per ogni file in un processo separato, che può essere eseguito in parallelo. Inoltre files non modificati dall'ultima compilazione possono essere riutilizzati senza ricompilare (compilazione incrementale); tutto ciò porta ad un tempo di compilazione notevolmente rapido.
Come visto precedentemente la compilazione si divide in più fasi: parsing, check dei tipi, ottimizzazioni del linguaggio intermedio (SIL), backend LLVM: il parsing ed il check di tipi sono i meno dispendiosi; l'ottimizzazione del

SIL tipicamente richiede un terzo del tempo totale di compilazione, mentre gli altri due terzi sono impegnati dal backend LLVM che opera ottimizzazioni di basso livello e genera il codice.
Con WMO, dopo aver effettuato le ottimizzazioni sull'intero modulo nella fase di ottimizzazione del linguaggio intermedio, il modulo è separato di nuovo in parti multiple. I processi di LLVM processano le parti in thread multipli e inoltre evitano il reprocessing delle parti che non sono cambiate rispetto all'ultima build. In conclusione WMO è un modo veloce ed efficace per ottenere ottime prestazioni, senza l'obbligo di gestire il codice Swift in vari file in un modulo. Se le ottimizzazioni, come descritto precedentemente, riescono ad essere efficaci per buona parte del codice, si ottengono incrementi prestazionali fino a cinque volte rispetto alla compilazione a file singolo.

2.1.3 Utilizzo in iOS e frameworks Cocoa

Al momento l'utilizzo di Swift nei frameworks di supporto al sistema operativo e nell'SDK è molto limitato, poichè il linguaggio non ha ancora raggiunto una maturità tale da poter avere delle ABI (Application Binary Interface) stabili, in quanto ogni singola versione pubblicata fino ad ora ha modificato interfacce, dipendenze e sintassi, rendendo necessaria una revisione del codice già scritto per la versione precedente.
A runtime, i binari Swift interagiscono con le altre librerie e gli altri componenti attraverso le ABI, ovvero le specifiche alle quali i binari compilati indipendentemente devono conformarsi per per essere collegati ed eseguiti: queste entità devono conformarsi su come chiamare le funzioni, come i dati sono rapppresentati in memoria, dove sono salvati i metadata e come vanno acceduti.
Le ABI sono specifiche per ogni piattaforma, e sono influenzate sia dall'architettura che dal sistema operativo.
La maggior parte dei creatori di piattaforme definiscono uno standard di ABI usate per il linguaggio C e basate su linguaggi della famiglia dello stesso. Swift, per sua natura è molto differente dal C e quindi si rende necessario creare ABI speficiche.
Avere delle ABI stabili significa bloccarle per far si che future versioni del compilatore possano produrre binari che si conformino alla versione stabile delle stesse, e la loro chiusura tende ad essere definitiva per tutta la vita della piattaforma.
Avere ABI stabili ha effetti solamente sulle interfacce pubblicamente visibili e sui simboli; simboli usati iternamente, convenzioni e interfacce possono continuare ad essere modificati senza effetti distruttivi, per esempio una versione futura del compilatore è libera di modificare le convenzioni per le chiamate

di funzione interne, purchè le interfacce pubbliche non vengano modificate
Le decisioni sulle ABI avranno effetti ramificati ed a lungo termine e quindi potrebbero limitare i modi in cui il linguaggio evolverà in futuro, perciò la comunità open source sta prendendo tempo per definire nel modo migliore possibile la struttura delle stesse per Swift; si prevede che la versione 4 porterà la dichiarazione di stabilità per le ABI rendendolo di fatto un linguaggio maturo.

Capitolo 3

Confronto tra i linguaggi

3.0.1 Sintassi

Objective-C, in quanto lunguaggio basato su C, ha introdotto nuove parole chiave per differenziare i nuovi tipi da quelli C, utilizzando il simbolo @. Swift, in quanto linguaggio indipendente non applica alcuna distinzione.
Swift elimina inoltre le convenzioni utilizzate nei linguaggi di programmazione con qualche decade sulle spalle e non solo: non è necessario utilizzare il punto e virgola per terminare un blocco di codice, non sono necessarie le parentesi per le espressioni condizionali negli statement if/else.
Una differenza sostanziale rispetto ad Objective-C sono le chiamate di funzione, che utilizzano la sintassi puntuale invece di quella con le parentesi quadre. Gli argomenti di funzione inoltre utilizzano la più comune virgola rispetto alle parentesi quadre innestate

3.0.2 Manutenzione del codice sorgente

Swift è più semplice da mantenere
Objective-C non può evolvere senza attendere l'evoluzione del linguaggio C sottostante; quest'ultimo richiede al programmatore il mantenimento di due files separati per migliorare il tempo di compilazione e l'efficienza di esecuzione, problema che si ripercuote su Obj-C.
In Swift questo problema non si presenta e il compilatore riconosce dipendenze automaticamente e performa build incrementali, il tutto utilizzando un singolo file.

3.0.3 Tooling

I meccanismi ausiliari di aiuto alla programmazione, quali evidenziazione della sintassi ed i suggerimenti di Swift non sono ancora alla pari di quelli Objective-C, fatto che si rende evidente confrontando due files scritti nei due linguaggi in XCode. Inoltre, gli strumenti di refactoring non sono ancora disponibili per Swift.

3.0.4 Runtime

Il runtime di Objective-C è generalmente più robusto e permette meccanismi quali reflection e deep introspection di oggetti e tipi, che al momento non sono disponibili in Swift

3.0.5 Sicurezza

Un aspetto interessante di Objective-C è il comportamento dei puntatori (in particolare quelli nil). In questo linguaggio non accade nulla se si chiama un metodo con una variabile puntatore non inizializzata: questa linea di codice viene considerata una non-operazione (no-op). Questo comportamento può avere effetti impredicibili, poichè può provocare effetti collaterali sull'esecuzione del programma.
I tipi optional di Swift invece offrono la possibilità di avere una gestione chiara del tipo nil, e genera errori a tempo di compilazione. Questo permette di creare un ciclo di feedback molto ristretto per lo sviluppatore e fa si che si scriva codice con attenzione a questo tipo di problema.
Tipicamente, in Objective-C, se un valore viene ritornato da una funzione è compito dello sviluppatore documentare il comportamento del puntatore ritornato (utilizzando commenti e convenzioni di nome); questo non accade in Swift in quanto il tipo optional permette a priori di capire se il valore esiste o se ha la possibilità di essere nil.
Per offrire un comportamento predicibile Swift genera un crash a runtime se una variabile nil di tipo optional è utilizzata.

3.0.6 Tempi di scrittura del codice

Come già analizzato, il tempo di scrittura di una classe in Swift è notevolmente minore grazie al singolo file necessario per la dichiarazione e definizione della stessa.

Altre caratteristiche di Swift che permettono di risparmiare tempi di scrittura sono la concatenazione di stringhe tramite l'operatore + (operazione non possibile in Objective-C), oltre alla possibilità di interpolare stringhe senza dover utilizzare sintassi quali %s, %d, %@). Il sistema di tipi in Swift inoltre riduce la complessità degli statements grazie alla type inference, ovvero la capacità del compilatore di capire il tipo degli oggetti senza la necessità di esplicitarlo nel codice.

3.0.7 Namespaces nei progetti open source

Uno dei problemi di Objective-C è la mancanza di supporto formale ai namespaces, soluzione utilizzata per evitare collision di nome nei files.
Quando ciò accade in questo linguaggio si ha un errore a livello di linking.
Alcune convenzioni sono state utilizzate, come per esempio prefissi a due o tre lettere per differenziare il codice scritto da un programmatore rispetto ad un altro, per esempio nei progetti condivisi su github, contenenti frameworks.
Swift supporta i namespaces permettendo quindi l'esistenza dello stesso file in progetti multipli senza causare un errore nel building, poichè sono basati sul target che contiene il file; questo significa che il programmatore può differenziare le classi o valori utilizzando l'identificatore del namespace.
In termini pratici ciò significa che nella collaborazione in progetti open source è possibile creare files con lo stesso nome evitando comunque collisioni ed errori in compilazione.

3.0.8 Librerie dinamiche

Un aspetto che ha suscitato poco clamore ma che può apportare benefici notevoli nel lungo periodo sono le librerie dinamiche di Swift: queste sono pezzi di codice eseguibile che possono essere collegati da un'applicazione. Questa caratteristica permette alle applicazioni già pubblicate di avere aggiornamenti delle librerie col susseguirsi delle versioni del linguaggio.
Lo sviluppatore invia sullo store l'applicazione insieme alle librerie, digitalmente firmate per assicurarne l'integrità, questo significa che Swift può evolvere più velocemente di iOS stesso, poichè gli aggiornamenti della libreria possono essere inclusi direttamente in un aggiornamento dell'applicazione.

3.0.9 Open Sourcing

Swift è stato reso open source nel Dicembre del 2015, permettendo agli sviluppatori di influenzare il futuro del linguaggio; questo ha già portato a notevoli cambiamenti nella struttura dello stesso nel passaggio dalla versione 2.3 alla 3.0

3.1 Gestione della memoria

Esistono due modalità di gestione della memoria:

-MMR (Manual retain-release), dove lo sviluppatore gestisce esplicitamente la memoria, tenendo traccia degli oggetti instanziati. E' implementato tramite un modello chiamato Reference Counting, fornito dalla classe NSObject in congiunzione all'ambiente di runtime; è il metodo più obsoleto e più dispendioso in termini di tempo di sviluppo in quanto è un approccio prettamente manuale.

-ARC (Automatic reference counting), che utilizza lo stesso sistema di tracciamento degli oggetti di MMR, ma aggiunge automaticamente chiamate ai metodi di gestione della memoria a tempo di compilazione. Questo sistema permette di assicurare che gli oggetti abbiano vita il tempo necessario per il loro utilizzo e non oltre, poichè il compilatore genera in automatico anche i metodi di dealloc appropriati.
E' l'approccio moderno e più utilizzato della gestione della memoria in Objective-C e Swift.

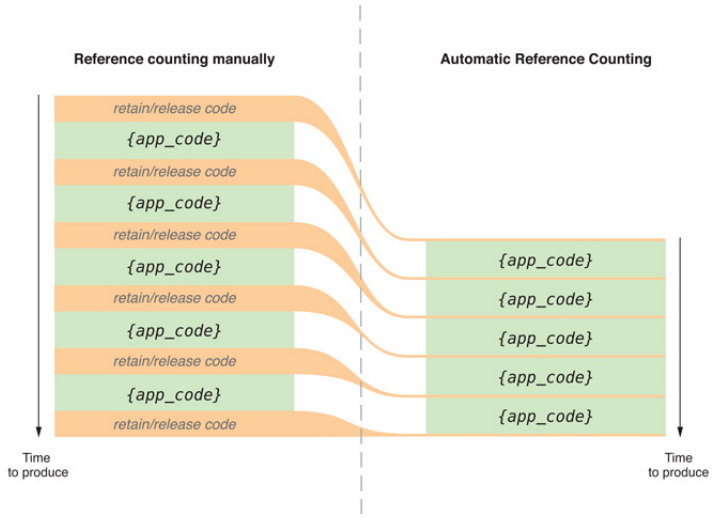

Figura 3.1: *Confronto tra MMC ed ARC relativo al tempo di creazione dei cicli di retain-release degli oggetti*

Il comportamento di ARC è però differente tra i due linguaggi:
In Swift il supporto è completo rispetto ai percorsi di codice procedurali ed object oriented; Objective-C invece supporta ARC solamente nell'utilizzo delle API Cocoa ed il codice object-oriented, questo significa che sarà ancora compito del programmatore gestire la memoria quando vengono utilizzate API come Core Graphics e altre di basso livello disponibili in iOS, creando il rischio di memory leaks.

3.2 Performance

Tester indipendenti hanno confrontato le performance dei due linguaggi su strutture dati standard quali Array/NSArray, Dictionary/NSDictionary e Set; i risultati sono molto variabili rispetto all'operazione effettuata sulla struttura dati.
L'approcio utilizzato prevede la preinizializzazione delle strutture dati con un numero fisso di elementi; è stata effettuata solamente una operazione sulla struttura dati, quindi è stata creata una nuova struttura con un nuovo stato iniziale ed è stata eseguita nuovamente l'operazione.
Sono stati considerati 500 stati differenti per ogni struttura dati, e le performance sono state calcolate su 10 iterazioni.
L'asse X mostra il numero di elementi nella struttura dati, l'asse Y il tempo medio di esecuzione dell'operazione.

Aggiunta di un elemento ad un array

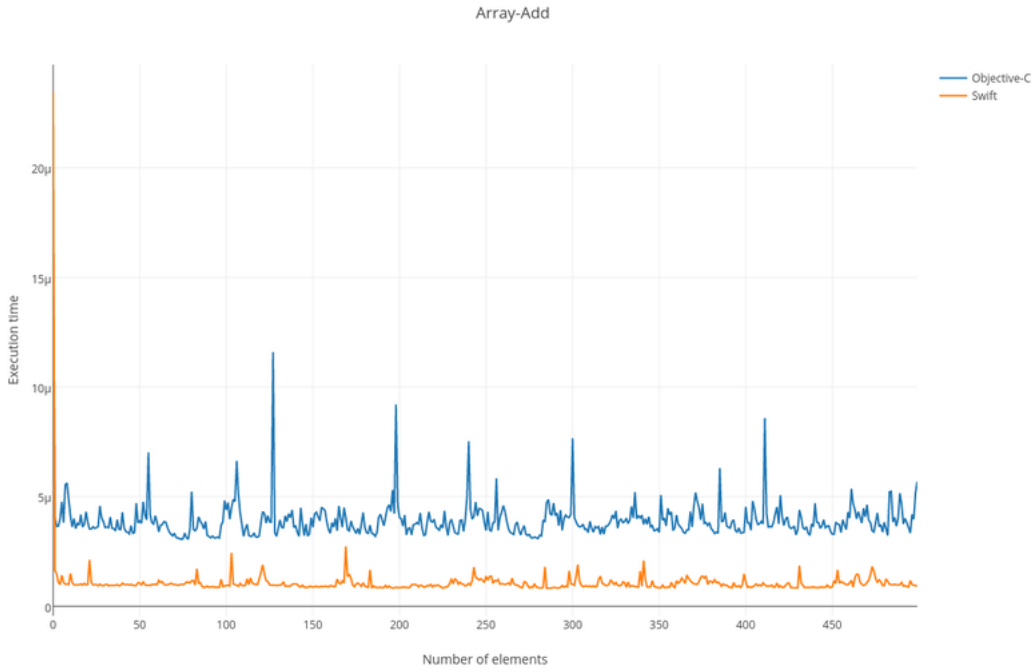

Figura 3.2: *L'aggiunta del primo elemento all'array dinamico in Swift è quattro volte più veloce rispetto ad Objective-C. L'operazione è effettuata in tempo costante per entrambi i linguaggi Per gli array contenenti già elementi l'operazione in Objective-C è più veloce di due volte rispetto a Swift*

Rimozione di un elemento in un array

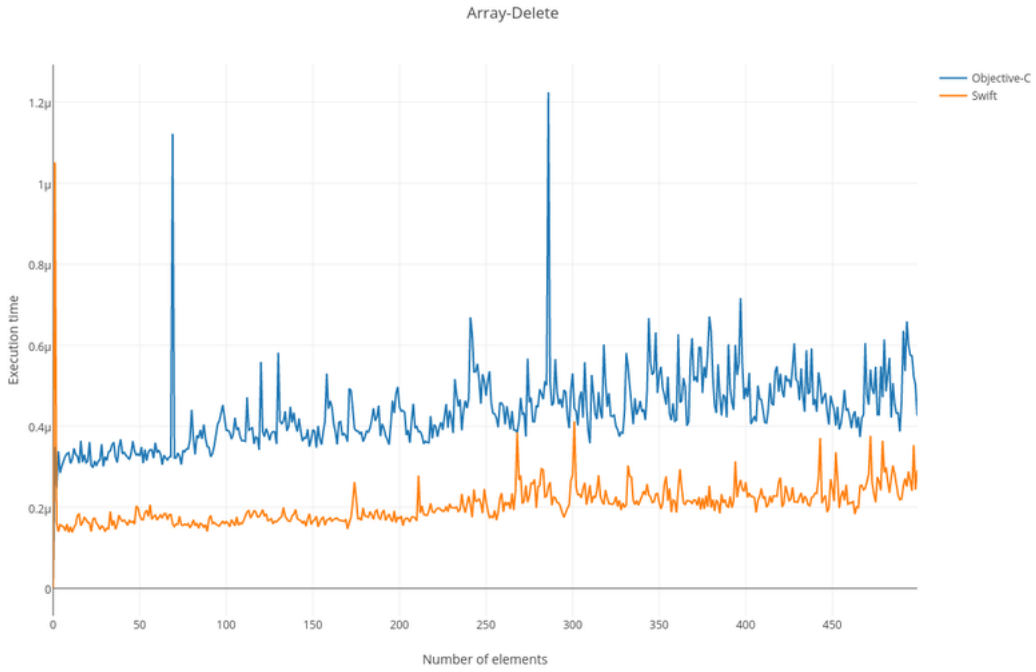

Figura 3.3: *C'è un significativo margine tra le performance dei due linguaggi per questa operazione: in Objective-C è inizialmente lineare, per poi passare a tempo costante alla fine; in Swift la complessità è lineare nell'intero intervallo*

Lettura di un elemento in un array

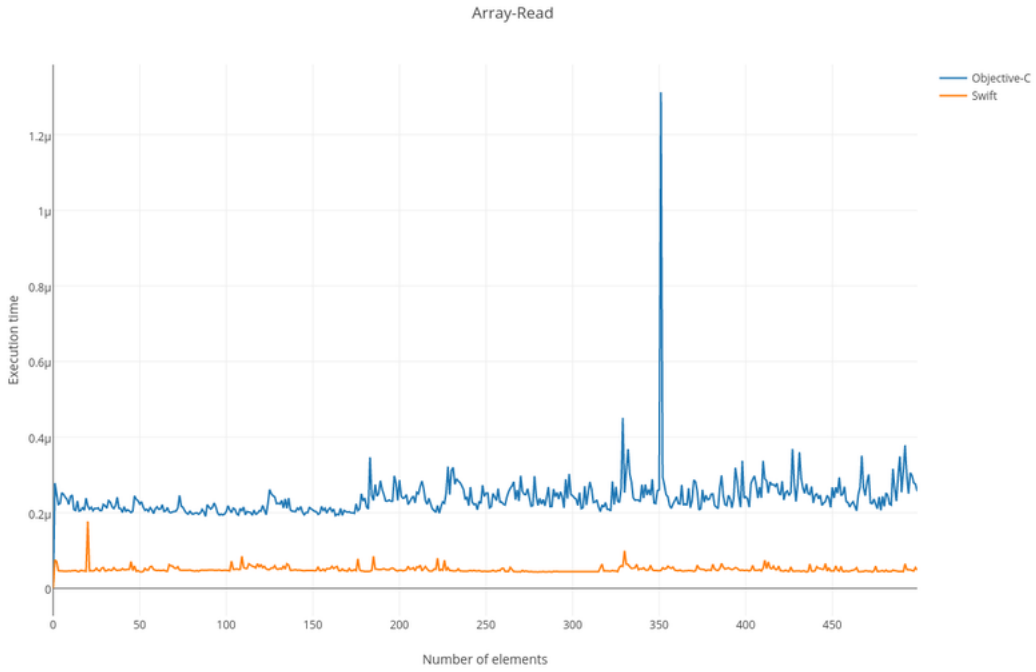

Figura 3.4: *Swift è 4-6 volte più veloce, entrambi i linguaggi performano in tempo costante, ma Swift risulta molto più stabile*

Ricerca di un elemento in un array

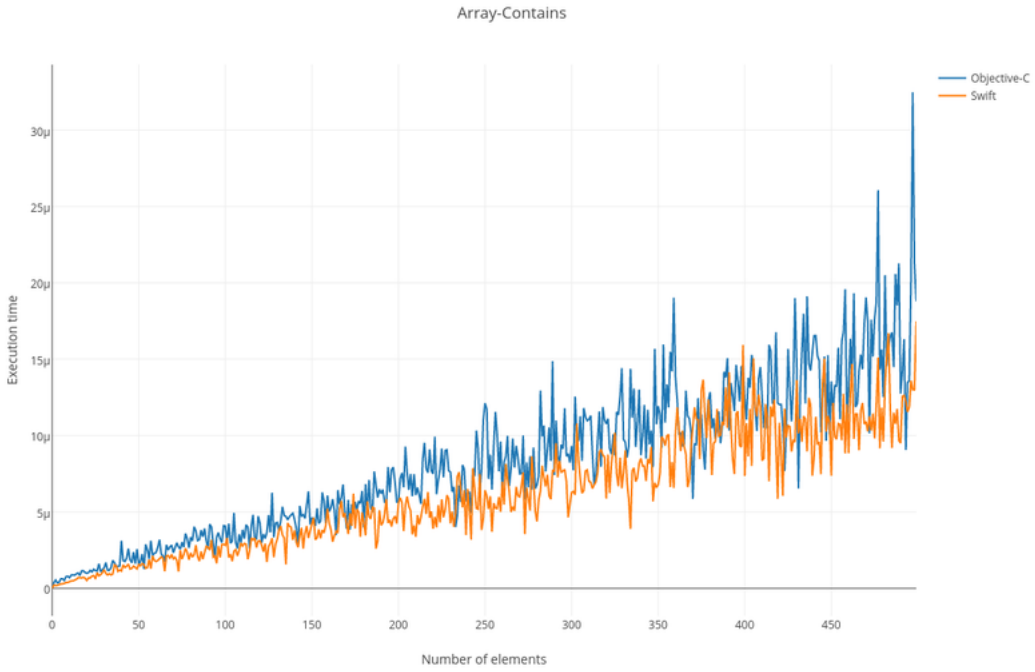

Figura 3.5: *I risultati per questa operazione vedono Swift leggermente più veloce. Complessità lineare per entrambi*

Aggiornamento di un elemento in un array

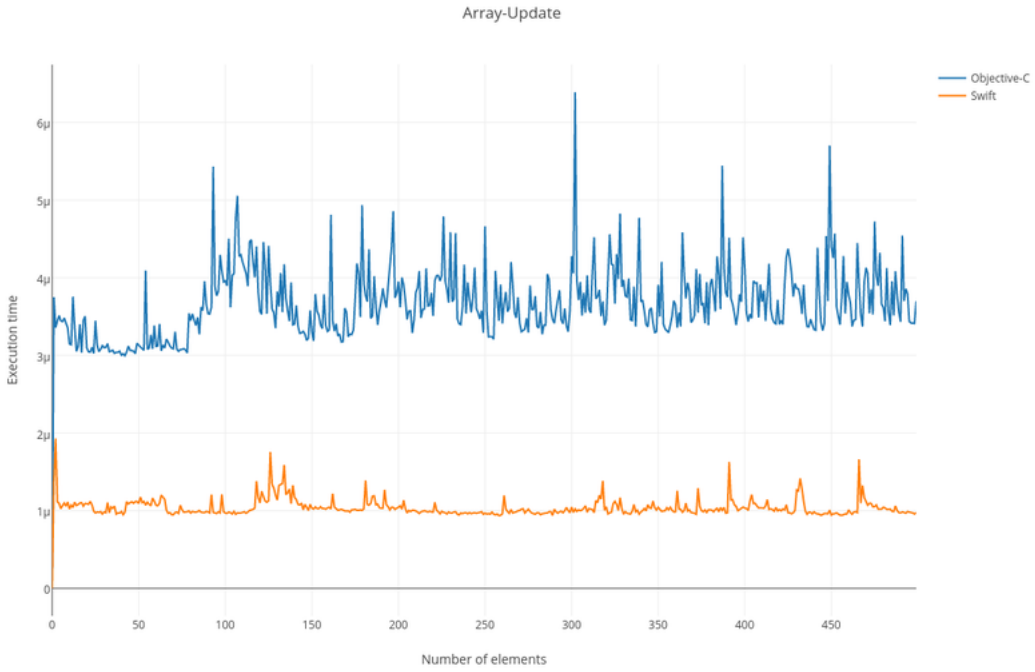

Figura 3.6: *Swift è 3-4 volte più veloce, ed ha complessità costante; Objective-C invece utilizza una funzione polinomiale e diventa costante solamente alla fine*

Aggiunta di un elemento ad un dizionario

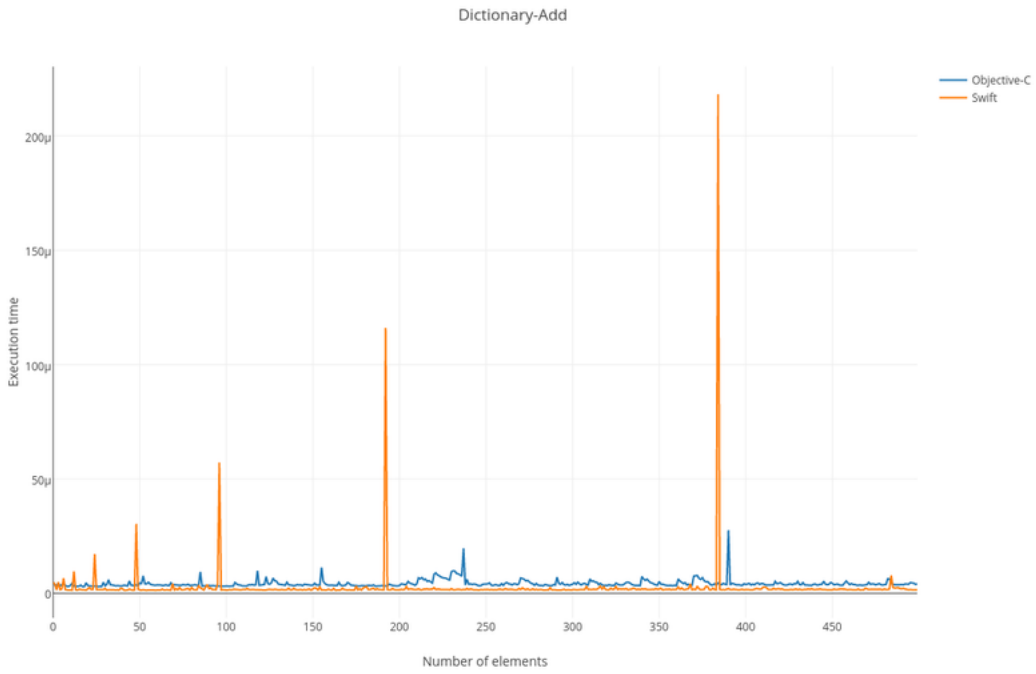

Figura 3.7: *Swift è 2-3 volte più veloce; si possono notare dei picchi regolari nel tempo di esecuzione, spiegabili con l'allocazione di memoria per i nuovi elementi, in quanto il dizionario aumenta di 2 volte e non linearmente*

Ricerca di un elemento in un dizionario

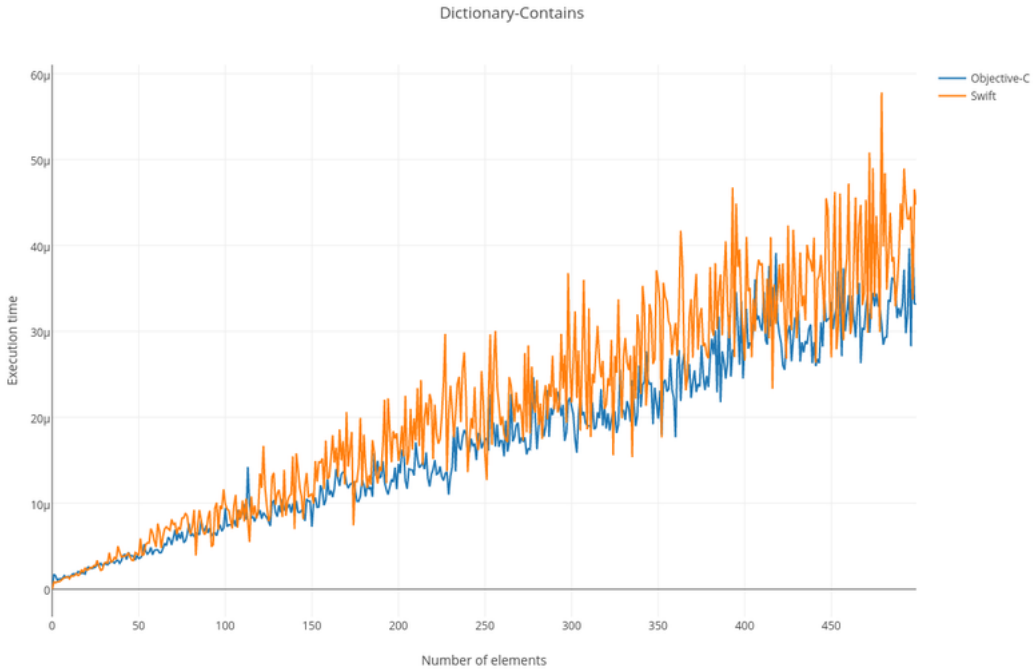

Figura 3.8: *I risultati sono simili alla ricerca in un Array: entrambi hanno complessità lineare, ma Swift ha performance minori*

Rimozione di un elemento da un dizionario

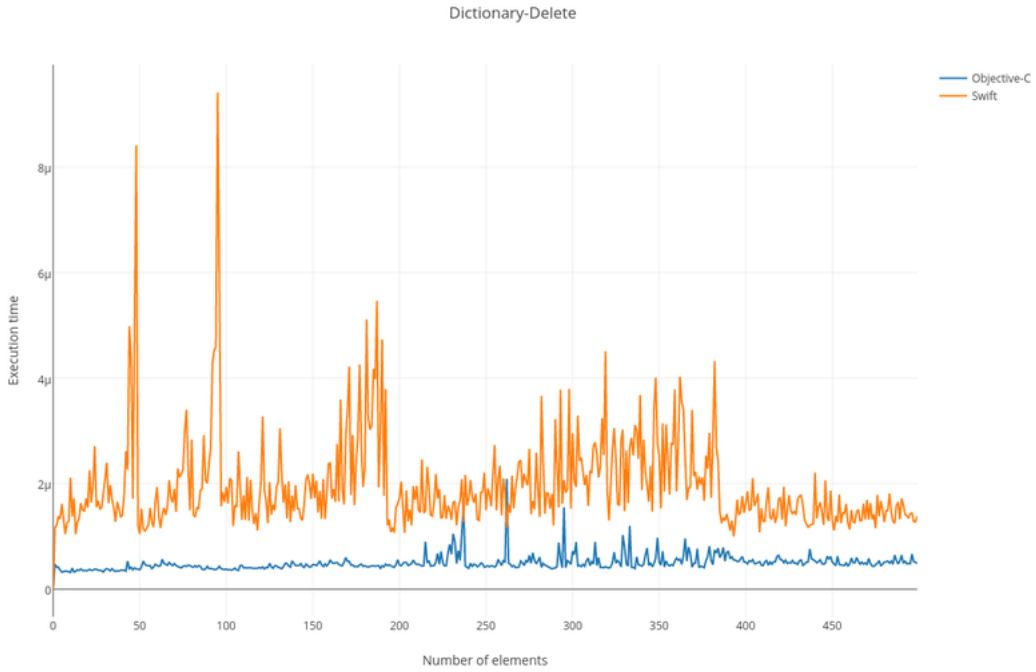

Figura 3.9: *Objective-C è 3-4 volte più veloce in questa operazione; con Swift otteniamo una curva con andamento ondulatorio, correlata al cambiamento dinamico della dimensione dell'array; entrambi hanno complessità costante*

Lettura di un elemento da un dizionario

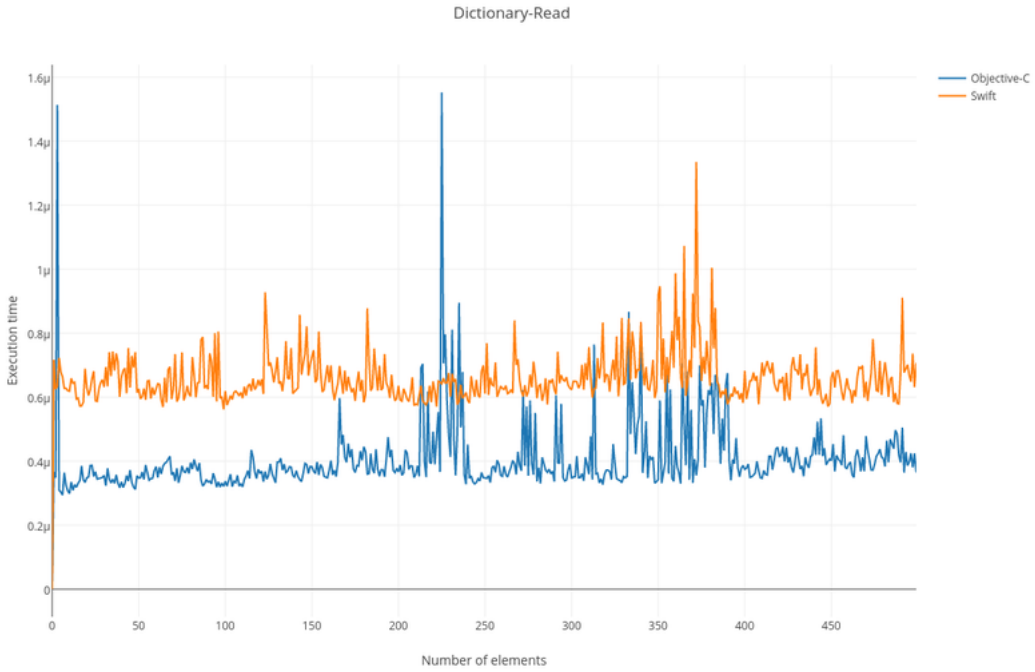

Figura 3.10: *Objective-C è due volte più veloce; la complessità è costante per entrambi i linguaggi*

Aggiornamento di un elemento di un dizionario

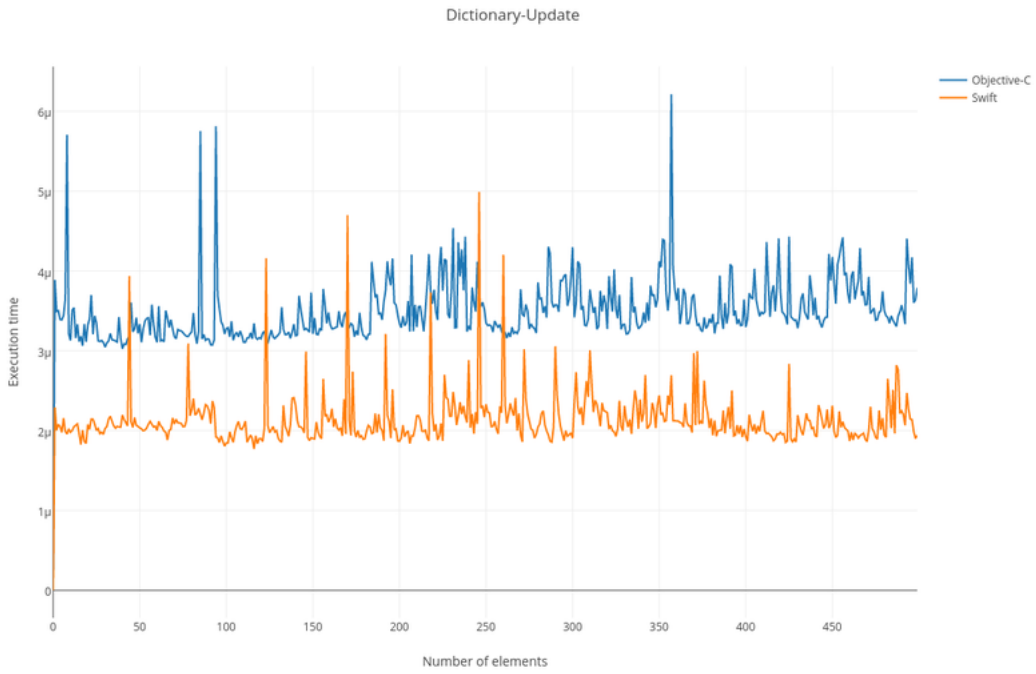

Figura 3.11: *Swift è due volte più veloce e la sua complessità è costante, mentre per Objective-C è lineare*

Aggiunta di un elemento ad un set

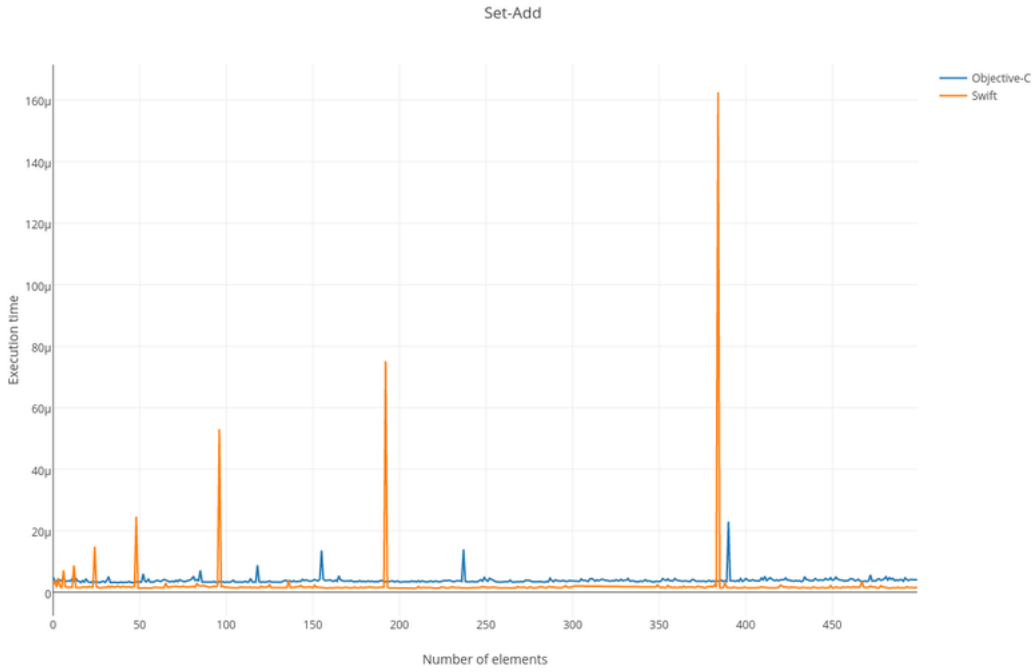

Figura 3.12: *Swift è due volte più veloce per questa operazione*

Ricerca di un elemento in un set

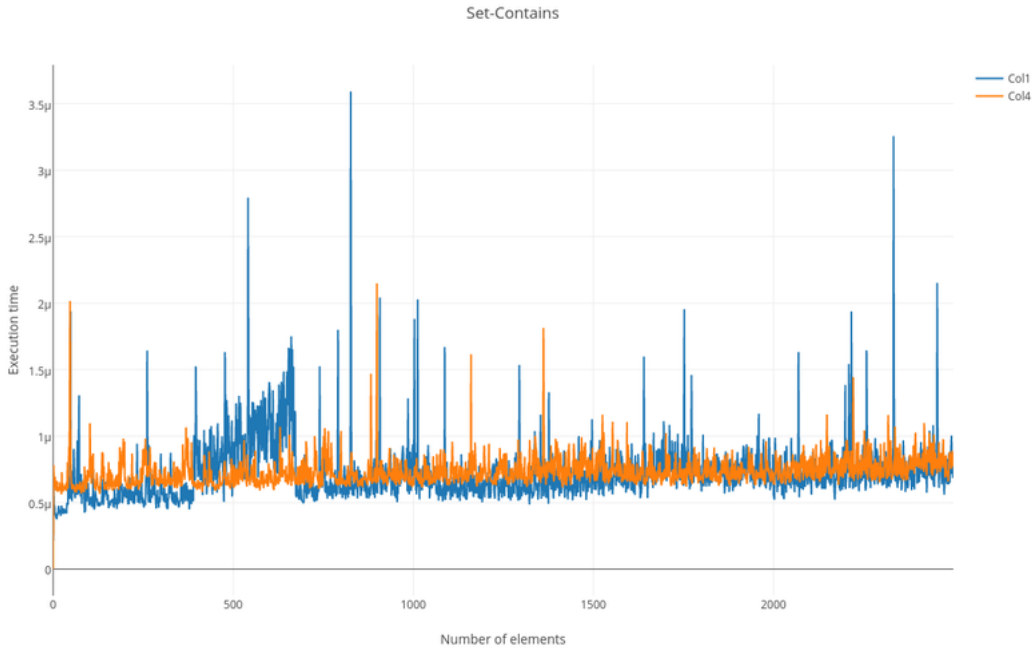

Figura 3.13: *Quando si considerano intervalli relativamente lunghi con 2500 elementi i grafici sono quasi conincidenti*

Rimozione di un elemento da un set

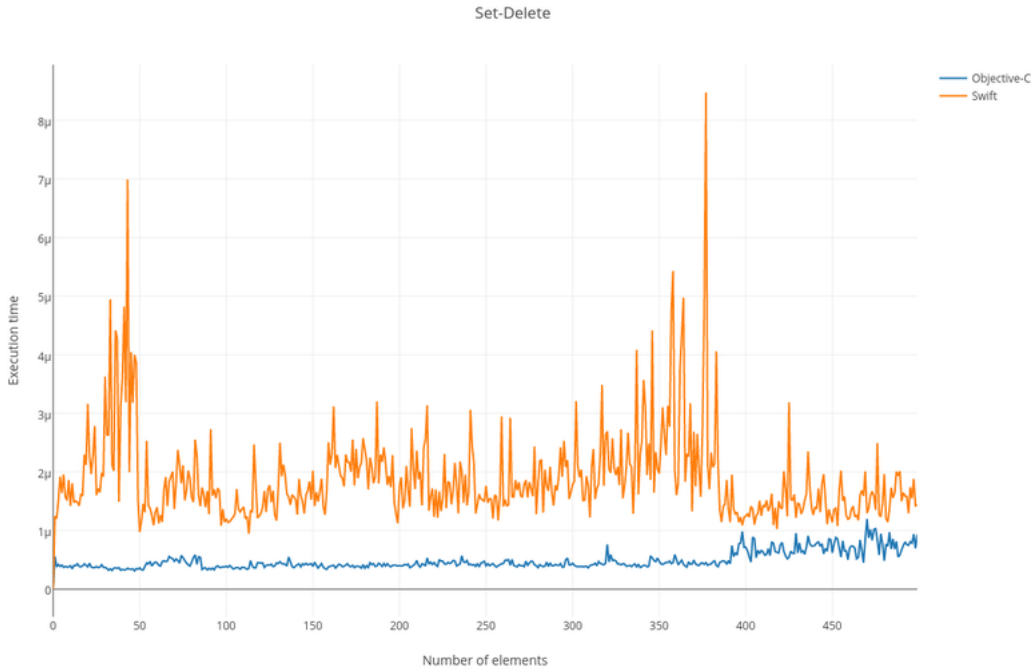

Figura 3.14: *Objective-C è 2-3 volte più veloce; entrambi i grafici mostrano complessità costante, ma Swift è più prono a fluttuazioni randomiche*

Conclusioni sulle performance

Le operazioni sugli Array sono dalle due alle quattro volte più performanti in rispetto a quelle sugli NSArray; per migliorare le prestazioni si dovrebbero preinizializzare le strutture dati con il numero di elementi massimo conosciuto in anticipo. Tutte le operazioni tranne la ricerca (contains) vengono eseguite in tempo costante; nonostante Swift gestisca le operazioni di inserzione in un dizionario e in un set in maniera più efficiente, le altre operazioni soffrono in prestazioni rispetto ad Objective-C.
Gli Array risultano la struttura dati da preferire in Swift per tutte le operazioni eccetto la ricerca se si hanno un gran numero di elementi; in questo caso un Set è preferibile.
Come si può notare, i grafici delle funzioni risultano particolari per un array in stile C, questo perchè a livello implementativo non è un vero array

C; Objective-C utilizza un complesso insieme di strutture dati che non sono Array nativi ma che espongono funzionalità da tale.

Capitolo 4

Il futuro della piattaforma iOS

4.1 Quale futuro per i due linguaggi?

Swift poco utilizzato al momento, persino da Apple Da intervista Lattner i due linguaggi andranno di pari passo per molto tempo Swift non ha ABI stabili Aggiungere foto o comunque menzionare l'apple tv, il watch, carplay

Conclusioni

TODO

Ringraziamenti

TODO

Bibliografia

[1] Objective-C
Publications, Books, Articles, Interviews, etc.
`http://virtualschool.edu/objectivec/`