

Spis treści

| | |
|--|-----------|
| 1. Wstęp | 3 |
| 2. Podobieństwa do innych języków programowania | 5 |
| 2.1. Podstawowe cechy | 5 |
| 2.2. Typy generyczne | 5 |
| 2.3. Typy wartościowe i referencyjne | 7 |
| 2.4. Domknięcia jako typy pierwszoklasowe | 8 |
| 2.5. Leniwość | 9 |
| 2.6. Elementy zaczerpnięte z języków funkcyjnych | 10 |
| 2.7. Zwięzłość składni | 12 |
| 2.8. Rozszerzenia typów | 12 |
| 3. Podobieństwa i różnice pomiędzy Swiftem i Objective-C | 15 |
| 3.1. Podobieństwa | 15 |
| 3.1.1. Swift i Objective-C jako języki programowania obiektowego . | 15 |
| 3.1.2. Zarządzanie pamięcią | 17 |
| 3.1.3. Biblioteki | 17 |
| 3.2. Różnice | 18 |
| 3.2.1. Składnia | 18 |
| 3.2.2. Zarządzanie pamięcią | 18 |
| 3.2.3. System typów | 19 |
| 3.2.4. Bezpieczeństwo | 19 |
| 4. Testy | 21 |

| | | |
|-----------|---|-----------|
| 4.1. | Założenia i platforma testowa | 21 |
| 4.2. | Wstawianie elementu do tablicy | 22 |
| 4.2.1. | Analiza działania | 22 |
| 4.3. | Rekurencyjne obliczanie liczby Fibonacciego | 25 |
| 4.3.1. | Analiza działania | 25 |
| 4.4. | Sortowanie bąbelkowe | 26 |
| 4.4.1. | Analiza działania | 26 |
| 4.5. | Budowanie binarnego drzewa poszukiwań | 30 |
| 4.5.1. | Analiza działania | 30 |
| 4.6. | Rodzaje wywołania metod | 34 |
| 4.6.1. | Analiza działania | 35 |
| 4.7. | Złożony algorytm | 38 |
| 4.7.1. | Analiza działania | 39 |
| 4.8. | Pozostałe testy | 42 |
| 4.8.1. | Zliczanie słów | 42 |
| 4.8.2. | Sito Eratostenesa | 42 |
| 4.8.3. | Zliczanie liter, słów i linii | 42 |
| 4.8.4. | Konkatenacja napisów | 42 |
| 4.8.5. | Histogram RGB | 43 |
| 4.8.6. | Szyfr RC4 | 43 |
| 4.8.7. | Wyniki pozostałych testów | 43 |
| 5. | Wnioski | 45 |
| 5.1. | Podsumowanie wyników | 45 |
| 5.2. | Wady i zalety Swifta na podstawie testów | 46 |
| 5.3. | Inspiracje dla Swift z innych języków programowania | 46 |

Rozdział 1.

Wstęp

Swift to język programowania stworzony przez firmę Apple na potrzeby tworzenia aplikacji na platformy iOS, Mac OS, tvOS i WatchOS. Jego pierwsza wersja została zapowiedziana 2 czerwca 2014 roku podczas konferencji Apple Worldwide Developers Conference (WWDC) i wydana 9 września 2014 roku.

Przed 2014 rokiem głównym językiem używanym do tworzenia oprogramowania na systemy operacyjne Apple był język Objective-C. Powstał on w 1983 roku jako nakładka język na C rozszerzająca go o elementy obiektowe wzorowane na Smalltalku. Głównym problemem Objective-C jest wysoki poziom trudności. Składnia tego języka różni się znacznie od składni innych popularnych języków obiektowych, jest dużo bardziej skomplikowana i mniej czytelna. Również wydajność nie jest najlepsza. W celu uzyskania szybko działającego kodu programiści często wykorzystują fakt, że Objective-C współpracuje z językiem C i używają go do zaimplementowania najważniejszych części aplikacji.

Swift ma za zadanie rozwiązać problemy, z którymi zmaga się Objective-C. Jego składnia jest dużo prostsza i bardziej podobna do innych obiektowych języków programowania. Zaadresowano również zwiększenie wydajności i bezpieczeństwa kodu. Swift otrzymał nowe funkcje pozwalające pisać kod szybciej, wydajniej i bezpieczniej. Dokładne porównanie obydwu języków znajduje się w rozdziale 3.

Swift zaczął bardzo szybko zyskiwać na popularności. Według ankiety przeprowadzanej co roku przez serwis **StackOverflow.com** już w 2017 roku nowy język był bardziej popularny od swojego poprzednika ¹. Należy również zwrócić uwagę na kategorię "Most Loved Languages", w której Swift rokrocznie zajmuje miejsce w pierwszej piątce.

3 grudnia 2015 roku kod źródłowy języka został udostępniony w serwisie GitHub. W tym samym momencie ogłoszono również wsparcie dla systemów z rodziny Linux. Te dwa wydarzenia pozwoliły na próby zastosowania tego języka w innych

¹<https://insights.stackoverflow.com/survey/2017#most-popular-technologies>

gałęziach IT innych niż aplikacje mobilne. W niedługim czasie powstało kilka ram do tworzenia aplikacji serwerowych, takich jak Kitura, Vapor czy Perfect. Firma IBM dodała również wsparcie dla Swifta w swoich serwisach chmurowych. Kolejnym potencjalnym zastosowaniem jest uczenie maszynowe. Pierwszą próbą w tym kierunku jest wersja platformy TensorFlow wykorzystująca Swift (Swift for TensorFlow).

Pomimo szybko rosnącej popularności Swifta, istnieje mało źródeł traktujących o wydajności nowego języka. W Internecie można znaleźć kilka stron i blogów zawierających testy wydajnościowe, takich jak np. "The Computer Language Benchmark Game"² czy blog firmy Yalantis³, żaden z autorów nie pokusił się jednak o bardziej wnikliwą analizę.

Z tego też powodu, głównym celem niniejszej pracy jest porównanie wydajności programów napisanych w Swift z Objective-C oraz analiza przyczyn lepszej (lub gorszej) wydajności nowego języka. Aby to osiągnąć, zaimplementowano w obydwu językach szereg testów wydajnościowych. Następnie za pomocą narzędzia profilującego Instruments dokonano analizy szybkości działania kodu w zależności od języka, w którym został napisany. Szczegółowe wnioski znajdują się w rozdziale 4., natomiast podsumowanie zostało przedstawione w rozdziale 5.

²<https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/swift.html>

³<https://yalantis.com/blog/is-swift-faster-than-objective-c>

Rozdział 2.

Podobieństwa do innych języków programowania

2.1. Podstawowe cechy

Swift to wieloparadygmatowy język programowania łączący pomysły znane z innych popularnych języków, takich jak: Objective-C, C#, Rust, Haskell czy Ruby. Podobnie jak C#, pozwala na tworzenie struktur (typów wartościowych), klas (typów referencyjnych) i typów wyliczeniowych. Wspiera również dziedziczenie (ale nie wielokrotne), definiowanie protokołów (odpowiednik interfejsów z C# czy Java) oraz polimorfizm parametryczny (typy generyczne), nie pozwala natomiast na definiowanie klas abstrakcyjnych, zachęcając tym samym programistów do szerokiego stosowania interfejsów.

2.2. Typy generyczne

Swift wspiera dwie podstawowe koncepcje generyczności:

- klasy, struktury, typy wyliczeniowe oraz funkcje z parametrami typu (ang. *generics*)
- protokoły z powiązаныmi typami (ang. *associated types*)

Klasy (struktury, funkcje) ze zmiennymi typu to pomysł dobrze znany z większości popularnych języków programowania obiektowego, takich jak C# czy Java. W momencie definiowania klasy programista ma możliwość zdefiniowania zmiennych przebiegających przestrzeń typów używanych w definiowanej klasie. Dodatkowo Swift oferuje kilka bardziej zaawansowanych mechanizmów związanych ze zmiennymi typu:

- możliwość dodania ograniczeń na typy, po których przebiega zmienna, np. zmienna może być tylko typem implementującym dany protokół lub dziedziczącym po danej klasie
- automatyczna inferencja typów parametrów generycznych
- możliwość nadawania aliasów funkcjom i typom generycznym

Przykład użycia typów generycznych ilustruje Listing 1.

```
class Stack<T> {
    var stack: Array<T> = []
    func push(object: T) {
        stack.append(object)
    }
    func pop() -> T {
        return stack.removeLast()
    }
}

class PopManyStack<T> : Stack<T> {
    func popN(n: Int) -> [T] {
        let lastN = stack.suffix(n)
        stack.removeLast(n)
        return Array(lastN)
    }
}
```

Listing 1: Przykład klasy generycznej i klasy pochodnej w Swift

W odróżnieniu od klas, struktur i funkcji, protokoły nie wspierają generycznych parametrów typu. Zamiast tego, protokoły posiadają mechanizm typów powiązanych (ang. *associated types*), wzorowany na znanym np. ze Scali mechanizmie abstrakcyjnych pól typu (ang. *abstract type members*). Pozwala on na zdefiniowanie w protokole zmiennej typu, która zostanie ukonkretniona dopiero przez klasę implementującą dany protokół. Główną zaletą tego rozwiązania jest ukrycie typu podstawionego pod zmienną typu przed programistą używającym klasy implementującej dany protokół - typ podstawiony pod zmienną jest częścią implementacji i nie musi być jawnie podawany podczas tworzenia obiektu implementującego protokół. Przykład użycia protokołu z parametrami typu prezentuje Listing 2.

```
import UIKit

protocol ViewDecorator {
    associatedtype ViewType: UIView
    func decorate(view: ViewType)
}
```

```

class ImageViewDecorator: ViewDecorator {
    typealias ViewType = UIImageView

    func decorate(view: UIImageView) {
        view.backgroundColor = UIColor.black
        view.layer.cornerRadius = 5.0
        // ... więcej ustawień
    }
}

class LabelDecorator: ViewDecorator {
    typealias ViewType = UILabel

    func decorate(view: UILabel) {
        view.font = UIFont.systemFont(ofSize: 20.0)
        // ...
    }
}

let decorator1 = ImageViewDecorator() // nie trzeba podawać typu generycznego,
                                     // implementacja klasy wskazuje, czym jest ViewType
let decorator2 = LabelDecorator()
decorator1.decorate(view: UIImageView()) // system typów pilnuje,
                                     // aby view było typu UIImageView
decorator2.decorate(view: UILabel())

```

Listing 2: Przykład protokołu z typem powiązany w Swift

2.3. Typy wartościowe i referencyjne

Podobnie jak w języku C#, typy w Swiftcie można podzielić na dwie grupy:

- typy wartościowe (ang. *value types*)
- typy referencyjne (ang. *reference types*)

Typy wartościowe to typy, które tworzą nowe instancje obiektów podczas przypisywania do zmiennej lub przekazywania do funkcji. Innymi słowy, każda instancja posiada swoją własną kopię danych, obiekty takie nie dzielą ze sobą stanu, przez co są łatwiejsze w zrozumieniu i bezpieczniejsze przy pracy z wieloma wątkami. Jeśli zmienna typu wartościowego zostanie zadeklarowana jako stała, cały obiekt, łącznie ze wszystkimi polami nie może zostać zmieniony. Typami wartościowymi w Swiftcie są:

- struktury

- typy wyliczeniowe
- krotki

Typy referencyjne to typy, których obiekty dzielą pomiędzy sobą te same dane, a podczas przypisywania lub przekazywania do funkcji tworzona jest tylko nowa referencja do tych samych danych. Zmienne typu referencyjnego zadeklarowane jako stałe zapewniają jedynie stałość referencji, jednak dane przypisane do zmiennej mogą być bez dowolnie zmieniane. Typami referencyjnymi w Swiftcie są tylko klasy.

```
// struktura - typ wartościowy
struct UserInfo {
    let name: String
    let identifier: Int
}
// typ wyliczeniowy
enum ScreenResolution {
    case SD
    case HD
    case FullHD
}
// protokół - obiekt definiujący interfejs klasy/struktury go implementującej
protocol UserInfoDrawer {
    func drawUserInfo(info: UserInfo) -> Void
}
// klasa - typ referencyjny
class TerminalUserInfoDrawer: UserInfoDrawer {
    var screenResolution: ScreenResolution = .FullHD

    func drawUserInfo(info: UserInfo) -> Void {
        // .. implementacja wyświetlania informacji o użytkowniku
    }
}
```

Listing 3: Przykładowe definicje podstawowych obiektów w Swift: struktury, klasy, protokołu i typu wyliczeniowego

2.4. Domknięcia jako typy pierwszoklasowe

Podobnie jak w językach funkcyjnych i w większości nowoczesnych języków programowania obiektowego, domknięcia w Swiftcie są typem pierwszoklasowym (ang. *first-class citizen*), tzn:

- mogą być przechowywane w zmiennych i stanowić elementy struktur danych
- mogą być podawane jako parametry wywołania funkcji i metod

- mogą być zwracane przez funkcje i metody

```
// domknięcie przyjmujące dwa obiekty typu Int i zwracające obiekt typu Int
let addTwoInts: ((Int, Int) -> Int) = { (a, b) in a + b }

// wywołanie domknięcia przypisanego do zmiennej `addTwoInts`
let result = addTwoInts(5, 10)
```

Listing 4: Przykład użycia domknięcia w Swift

2.5. Leniwość

Swift jest domyślnie językiem z ewaluacją gorliwą, autorzy zaimplementowali jednak dwa rozwiązania pozwalające w podstawowym stopniu na wspieranie leniwych obliczeń. Po pierwsze, w Swiftcie, podobnie jak w C#, istnieje możliwość leniwej inicjalizacji obiektów. O ile jednak w C# mechanizm ten polega na użyciu klasy *Lazy* z biblioteki standardowej, o tyle w Swiftcie jest on zaszyty w samym języku - służy do tego słowo kluczowe *lazy*. Drugim rozwiązaniem są leniwe struktury danych, których implementacja opiera się na znanych również z języka C# czy Java generatorach.

```
typealias BigInt = Int

class DataContainer {
    // zmienna tworzona leniwie, dopiero podczas jej pierwszego użycia
    lazy var bigData = BigInt()

    // .. reszta ciała klasy
}

let container = DataContainer()
print(container.bigData) // obiekt bigData zostanie utworzony dopiero w tym momencie

// sekwencja liczb Fibbonacciego generowana leniwie
class Fibonacci: LazySequenceProtocol {
    public func makeIterator() -> FibonacciIterator {
        return FibonacciIterator()
    }
}

// generator liczb Fibbonacciego
class FibonacciIterator: IteratorProtocol {
    private var first = 0
    private var second = 1

    public func next() -> Int? {
        let next = first + second
```

```

        first = second
        second = next
        return next
    }
}

let evenFibonacci = Fibonacci().filter { $0 % 2 == 0 }
var iterator = evenFibonacci.makeIterator()

for i in 1...5 {
    print(iterator.next()!)
}

```

Listing 5: Przykład deklaracji leniwej zmiennej i leniwej sekwencji w Swift

2.6. Elementy zaczerpnięte z języków funkcyjnych

Pomimo tego, że Swift był projektowany głównie jako język programowania obiektowego, jego twórcy skupili dużą część swojej uwagi na elementach powiązanych z programowaniem funkcyjnym, które mogłyby pomóc programistom pisać bezpieczniejszy i bardziej czytelny kod obiektowy. Najważniejsze z nich to:

- Typy wyliczeniowe z wartościami powiązanymi (ang. *associated values*), które pozwalają na definiowanie typów podobnych do algebraicznych typów danych (ang. *Algebraic data types* znanych z programowania funkcyjnego (zob. Listing 6)).

```

indirect enum Tree<Value> {
    case Empty
    case Node(Tree<Value>, Value, Tree<Value>)
}

let intTree = Tree.Node(
    .Node(.Empty, 1, .Empty),
    2,
    .Empty
)

```

Listing 6: Implementacja drzewa binarnego w Swift

- Dzięki zwięzłej i eleganckiej składni oraz potraktowaniu domknięć na równi klasami i strukturami, Swift oferuje bardzo dobre wsparcie dla funkcji wyższego rzędu. Funkcje wyższego rzędu są też często używane w bibliotece standardowej, np. kolekcje danych posiadają najczęściej używane funkcje służące do manipulowania nimi, takie jak `filter`, `map`, `reduce` czy `flatMap`.

- Autorzy Swifta postawili bardzo duży nacisk na niemutowalność (*TODO: lepsze słowo?*) danych, co przejawia się w całej składni języka, np. dostępne jest słowo odrębne słowo kluczowe `let` służące do deklarowania stałych, parametry przekazywane do funkcji są domyślnie stałymi, użycie typów wartościowych jest preferowane nad użyciem klas (również w bibliotece standardowej) itp.
- Swift posiada zaawansowany mechanizm *pattern matchingu*, który można wykorzystać do dopasowywania typów wyliczeniowych, krotek i wyrażeń. Tak jak w wielu językach funkcyjnych, *pattern matching* w Swifcie jest wyczerpujący (ang. *exhaustive*), co oznacza, że każda wartość, która może pojawić się podczas dopasowywania musi zostać obsłużona.
- Aby uniknąć problemów z wartością `nil`, w języku Swift każda zmienna musi zostać zainicjalizowana już w momencie deklaracji. Jeśli programista chce celowo stworzyć zmienną mogącą przyjmować wartość `nil`, powinien użyć typu `Optional<T>`, który w swojej konstrukcji jest bardzo podobny do monady `Maybe` znanej z Haskella. Istnieje nawet mechanizm zwany *optional chaining*, który zachowuje się tak, jak operacja `>>=` dla monady `Maybe`.

```
// Implementacja typu Optional z biblioteki standardowej
public enum Optional<Wrapped> {
    case none
    case some(Wrapped)
}

// Monada Maybe w Haskellu
// data Maybe a = Nothing | Just a

struct Address {
    let city: String? // String? to cukier syntaktyczny dla typu Optional<String>
    let postalCode: String?
}

struct User {
    let name: String
    let address: Address?
}

let sampleUser: User? = User(
    name: "Jan Kowalski",
    address: Address(city: "Wrocław", postalCode: "50-500")
)

// Przykład użycia optional chaining - stała city ma typ Optional<String>
let city = sampleUser?.address?.city
```

Listing 7: Typ `Optional` i mechanizm *optional chaining*

2.7. Zwięzłość składni

Jednym z największych problemów podczas programowania w Objective-C była słaba czytelność kodu i bardzo rozwlekła składnia. Dlatego podczas projektowania Swifta inżynierowie Apple mocno wzorowali się na językach znanych ze swojej zwięzłości i łatwości czytania, takich jak Python czy Ruby. Zrezygnowano z plików nagłówkowych, wprowadzono dużo cukru syntaktycznego dla najczęściej stosowanych konstrukcji (jak np. operator `T?` dla typu `Optional<T>`), wprowadzono domyślną inferencję typów. Rysunek 8 pokazuje różnice pomiędzy kodem napisanym w Objective-C, a równoważnym kodem w Swift.

```
// Objective-C
if (myDelegate != nil) {
    if ([myDelegate respondsToSelector:
        @selector(scrollViewDidScroll:)]) {
        [myDelegate scrollViewDidScroll:myScrollView];
    }
}

// Swift
myDelegate?.scrollViewDidScroll?(myScrollView)
```

Listing 8: Przykładowy kod ilustrujący różnice w zwięzłości i czytelności Objective-C (na górze) i Swift (na dole). *WWDC Keynote 2014*

2.8. Rozszerzenia typów

Jedną z rzadziej spotykanych w językach statycznie typowanych językach programowania funkcjonalności jest możliwość rozszerzania istniejących już typów. Co prawda już w Objective-C programista miał moliwość stworzenia kategorii (ang. *category*), ale pozwalała ona tylko na dodawanie nowych funkcji, nie można było natomiast definiować nowych właściwości, konstruktorów ani typów zagnieżdżonych. Dlatego w Swiftcie zostały zaimplementowane rozszerzenia (ang. *extensions*), które pozwalają na:

- dodawanie nowych właściwości obliczanych (ang. *computed properties*)
- definiowanie nowych metod instancji i metod typu
- definiowanie nowych inicjalizatorów
- definiowanie i używanie typów zagnieżdżonych
- implementowanie metod protokołów

```
// Rozszerzenie dla typu String z biblioteki standardowej
extension String {
    func isEmpty() -> Bool {
        return self != ""
    }
}
```

Listing 9: Przykład rozszerzenia w Swift

Rozdział 3.

Podobieństwa i różnice pomiędzy Swiftem i Objective-C

3.1. Podobieństwa

3.1.1. Swift i Objective-C jako języki programowania obiektowego

Zarówno Swift, jak i Objective-C są głównie językami programowania obiektowego. Oba języki pozwalają na definiowanie własnych typów (klasy i typy wyliczeniowe w Objective-C, struktury, klasy oraz typy wyliczeniowe w Swiftcie), wspierają dziedziczenie i polimorfizm. Dzięki modyfikatorom dostępu oraz protokołom umożliwiają również enkapsulację implementacji i opisywanie zachowań za pomocą abstrakcyjnych typów danych.

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

// Definicja typu Book
@interface Book: NSObject
@property (copy, nonatomic, readonly) NSString *title;
@property (copy, nonatomic, readonly) NSString *author;
@property (copy, nonatomic, readonly) NSNumber *numPages;
- (instancetype)initWithTitle:(NSString *)title author:(NSString *)author
  numPages:(NSNumber *)numPages;
@end

@implementation Book
- (instancetype)initWithTitle:(NSString *)title author:(NSString *)author
  numPages:(NSNumber *)numPages {
    if (self = [super init]) {
        _title = title;
        _author = author;
        _numPages = numPages;
    }
    return self;
}
```

```

}
@end

// Definicja interfejsu BookPrinter
@protocol BookPrinter
- (void)printBook:(Book *)user;
@end

// Definicja klasy ConsoleBookPrinter implementującego protokół BookPrinter
@interface ConsoleBookPrinter: NSObject <BookPrinter>
@end

@implementation ConsoleBookPrinter
- (void)printBook:(Book *)book {
    NSLog(@"-----");
    NSLog(@"> Tytuł: %@", book.title);
    NSLog(@"> Autor: %@", book.author);
    NSLog(@"> Ilość stron: %@", book.numPages);
    NSLog(@"-----");
}
@end

int main (int argc, const char * argv[])
{
    Book* book = [[Book alloc] initWithTitle: @"Sztuka programowania"
                                             author: @"Donald Knuth"
                                             numPages: @(2338)];

    id<BookPrinter> printer = [ConsoleBookPrinter new];
    [printer printBook: book];
    return 0;
}

```

Listing 10: Przykład kodu obiektowego w Objective-C

```

struct Book {
    let title: String
    let author: String
    let numPages: Int
}

protocol BookPrinter {
    func printBook(_ book: Book)
}

class ConsoleBookPrinter: BookPrinter {
    func printBook(_ book: Book) {
        print("-----");
        print("> Tytuł: \(book.title)");
        print("> Autor: \(book.author)");
        print("> Ilość stron: \(book.numPages)");
        print("-----");
    }
}

```



```
}  
}  
  
let book = Book(title: "Sztuka programowania", author: "Donald Knuth", numPages: 2338)  
let printer: BookPrinter = ConsoleBookPrinter()  
printer.printBook(book)
```

Listing 11: Analogiczny kod napisany w Swiftcie

3.1.2. Zarządzanie pamięcią

TODO: *Może warto dodać, że ARC został wprowadzony ze względu na Swifta?* W początkowych wersjach systemu Mac OS i iOS zarządzanie pamięcią było w pełni manualne - co prawda obiekty (a właściwie wskaźniki do nich) posiadały liczniki referencji, jednak programista musiał sam zadbać o zarządzanie nimi. Przełom nastąpił w roku 2011, kiedy to do Objective-C zostało dodane automatyczne zliczanie referencji (ang. *automatic reference counting*, w skrócie: ARC).

Automatyczne zliczanie referencji to jedna z najprostszych metod zarządzania pamięcią, odciążająca programistę z obowiązku jawnego inkrementowania i dekrementowania liczników referencji. Użycie ARC w Objective-C powoduje wygenerowanie kodu, który zwiększa licznik referencji w momencie, gdy nowa referencja do obiektu zostaje utworzona (np. inicjalizacja, przypisanie, przekazania obiektu w parametrze) oraz zmniejsza go, w momencie usunięcia referencji. Dzięki temu programista nie musi manualnie używać funkcji `retain` i `release`, jak to miało miejsce w poprzednich wersjach Objective-C. Pozwala to na zapobiegnięcie wielu błędom, takim jak: wycieki pamięci, wielokrotne zwalnianie pamięci czy odwoływanie się do wcześniej zwolnionej pamięci. Jednocześnie, użycie ARC nie wprowadza niedeterminizmu, co ma miejsce w przypadku użycia automatycznego odśmiecania pamięci (ang. *garbage collector*) oraz ma znikomy wpływ na wydajność działania aplikacji.

ARC jest również metodą zarządzania pamięcią zaimplementowaną w Swiftcie. Główną różnicą jest jednak sposób implementacji. W Objective-C, ARC jest rozszerzeniem języka, operującym się głównie na preprocesorze i generowaniu kodu odpowiedzialnego za zliczanie referencji. W Swiftcie natomiast, ARC jest jego podstawową cechą, posiadającą odrębną składnię i wsparcie ze strony środowiska uruchomieniowego i kompilatora.

3.1.3. Biblioteki

Jedną z cech, dla której Swifta tak szybko zdobywa popularność jest możliwość wywoływania kodu Objective-C z poziomu kodu swiftowego i na odwrót (ang. *interoperability*). Z tego względu prawie wszystkie biblioteki standardowe posiadają bardzo podobne interfejsy i są dostępne w obu językach. Oba języki posiadają też

możliwość wywoływania kodu pisanego w języku C, dlatego też wiele z zaawansowanych bibliotek, jak np. CoreAudio czy GLKit (***TODO: źródło***) było dostępnych dla Swifta już od dnia jego prezentacji.

3.2. Różnice

3.2.1. Składnia

Objective-C to język, którego początki sięgają pierwszej połowy lat 80. Z tego powodu, niektóre jego cechy są w tym momencie uważane w środowisku developerów, za przestarzałe i niewygodne. Mając te cechy na uwadze, inżynierowie Apple opracowali nowy język z uproszczoną składnią i wieloma udogodnieniami pozwalającymi programistom w krótszym czasie pisać kod, który będzie czytelniejszy i łatwiejszy w utrzymaniu.

Pierwszym z udogodnień zaimplementowanych Swifta jest inferencja typów. W Objective-C każdy typ zmiennej musiał zostać jawnie napisany, co szczególnie w przypadku długich, złożonych typów (np. zawierających parametry generyczne) mogło być mocno uciążliwe. Kompilator Swifta natomiast stara się wywnioskować tak wiele informacji o typach, ile jest w stanie.

Po drugie, sama składnia języka jest dużo bardziej zwięzła i czytelna. W Swiftie usunięto konieczność pisania nawiasów przy warunkach instrukcji warunkowych i pętli, dodano możliwość oddzielania kolejnych instrukcji poprzez znak nowej linii (nie ma potrzeby pisania znaku średnika na końcu linii), zaimplementowano dużo prostszą obsługę stringów, dodano dużo czytelniejszą składnię dla funkcji wyższego rzędu, zrezygnowano z plików nagłówkowych (modyfikatory dostępu definiuje się podobnie jak w Javie czy C#). Dodatkowo, najczęściej używane struktury składniowe otrzymały prostsze formy w postaci cukru syntaktycznego, np:

- definicja zmiennej `var x: Int?` jest tożsama z `var x: Optional<Int>`
- konstrukcja `if let` jest cukrem syntaktycznym dla wyrażenia `switch`, wywołanego na obiekcie typu `Optional<T>`
- Swift 2.2 wprowadził cukier syntaktyczny dla selektorów (obiektów zawierających informacje pozwalające wywoływać na obiektach funkcje w runtime)
- pattern matching dla typów wyliczeniowych i krotek to bardzo mocno rozwinięty cukier syntaktyczny dla instrukcji warunkowych dla tych typów

3.2.2. Zarządzanie pamięcią

Pomimo, że zarówno Objective-C, jak i Swift korzystają z automatycznego zliczania referencji, jego implementacja jest diametralnie różna. W Objective-C ARC

został dodany jako jedno z rozszerzeń języka C, jego implementacja bazuje głównie na wykorzystaniu makr i preprocesora, który automatycznie wstawia kod odpowiedzialny za zarządzanie licznikiem referencji. W Swiftcie natomiast, ARC jest podstawowym elementem języka posiadającym odrębną składnię i słowa kluczowe.

3.2.3. System typów

Oba omawiane języki są językami statycznie typowanymi, jest jednak pomiędzy nimi kilka istotnych różnic:

- w Swiftcie w zasadzie nie występuje niejawne rzutowanie typów. Każde (nawet najprostsze, jak rzutowanie z typu całkowitoliczbowego na zmiennoprzecinkowy) musi być jawnie wywołane przez programistę. W Objective-C zasady rzutowania zostały odziedziczone z języka C i są w zasadzie takie same.
- sposób wywoływania metod w Objective-C został zaczerpnięty z języka Smalltalk i odbywa się poprzez wysyłanie wiadomości (ang. *message*) pomiędzy obiektami. Z tego względu rozwiązywanie, którą funkcję należy wywołać dzieje się w metodzie `objc_msgSend` w trakcie działania programu. W Swiftcie natomiast wywołania metod opierają się na wskaźnikach do metod oraz *Protocol Witness Table* (odpowiednik *vtable* z C++).
- w Objective-C istnieje klasa `NSObject`, która jest superklasą dla wszystkich innych klas. Swift nie posiada takiej superklasy, została ona zastąpiona protokołem `AnyObject`.

3.2.4. Bezpieczeństwo

Jednym z podstawowych założeń przyjętych przy tworzeniu Swifta było stworzenie języka, który będzie chronił programistę przed najczęstszymi błędami popełnianymi podczas pisania kodu. Najważniejsze mechanizmy służące temu celowi to:

- typ `Optional<T>` - typ ten chroni przed wszelkimi błędami związanymi z wartością `null`
- konieczność inicjalizacji obiektu - każdy obiekt musi zostać zainicjalizowany w trakcie zadeklarowania referencji, co zapobiega problemom z dostępem do jeszcze nie zainicjalizowanej pamięci
- generyczne struktury danych - Objective-C nie posiadało typów generycznych, przez co struktury danych mogły przechowywać wartości różnych typów, co z kolei powodowało konieczność sprawdzania typów. W Swiftcie struktury są silnie typowane i homogeniczne

- domyślna niemutowalność - w Swifcie istnieje wiele miejsc, gdzie obiekty są domyślnie niemutowalne, np przy przekazywaniu do funkcji
- preferowanie typów wartościowych nad referencyjne - typy wartościowe zapobiegają zmianom jednego obiektu z wielu miejsc, przez co czytelność i łatwość zrozumienia kodu jest dużo większa
- automatyczne sprawdzanie przekroczenia zakresu liczb całkowitych
- zmiany w składni, takie jak: obowiązkowe nawiasy `{...}` dla ciał funkcji warunkowych i pętli, obsługa wszystkich możliwych wartości dla typów wyliczeniowych, konieczność zwrócenia wartości w funkcjach, które deklarują zwracany typ, brak instrukcji `goto` itp.

Rozdział 4.

Testy

4.1. Założenia i platforma testowa

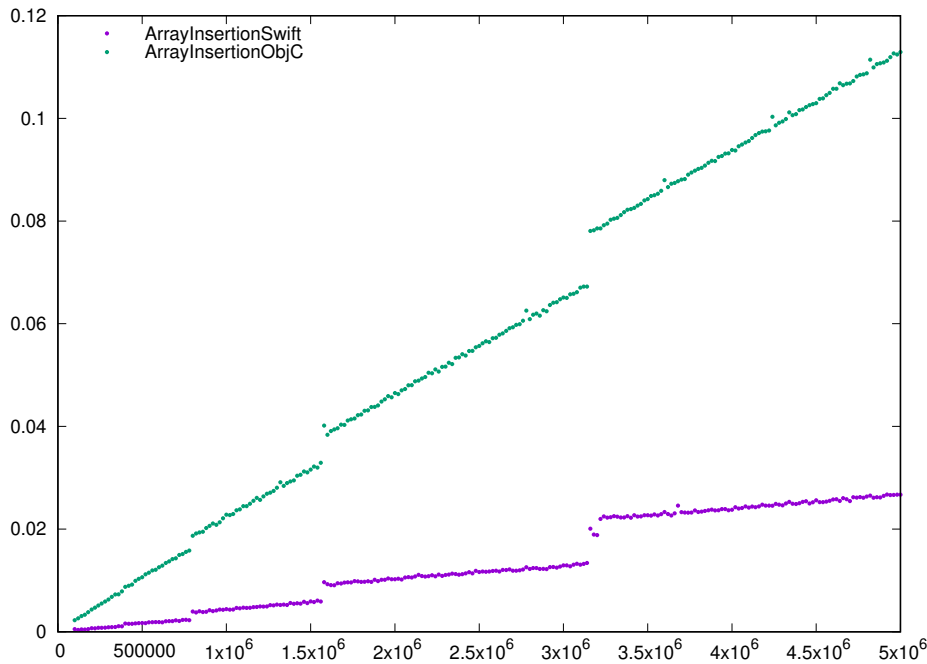
Kod testów, razem z platformą do ich urochomienia znajduje się w repozytorium dostępnym w serwisie GitHub pod adresem <https://github.com/vyeczorny/mgr>. Wszystkie testy opisane w tej pracy zostały wykonane na laptopie Macbook Pro o następującej specyfikacji:

- procesor: Intel Core I7-7700HQ 2,8 GHz
- pamięć RAM: 16 GB LPDDR3
- system operacyjny macOS High Sierra 10.13.1
- wersja Swift: 4.0.3 (swiftlang-900.0.74.1 clang-900.0.39.2)
- wersja LLVM: 9.0.0 (clang-900.0.39.2)

Głównym celem niniejszej pracy jest porównanie wydajności dwóch języków. Z tego powodu wszystkie implementacje w obrębie danego testu używają tego samego algorytmu do rozwiązania zadanego problemu. Również zakres wartości, dla których przeprowadzany jest test jest stały w obrębie danego testu. Dla każdej wartości n zostało przeprowadzone co najmniej 10 pomiarów, wyniki przedstawione na wykresach są średnimi tych pomiarów.

Każdy test został zaimplementowany w co najmniej jednej wersji dla każdego z języków. Analiza testu składa się z kilku części:

- opisu problemu przedstawionego w teście
- przedstawienia wyników w formie wykresu i/lub tablicy profilowania
- analizy wyników



Rysunek 4.1: Wykres czasu działania testu `ArrayInsertionTest` dla obu języków

- (opcjonalnie) dodatkowych testach sprawdzających wnioski wyciągnięta podczas analizy wyników

4.2. Wstawianie elementu do tablicy

Test polega na stworzeniu pustej tablicy, a następnie dodaniu do niej n elementów. Kod testów dla obu omawianych języków znajduje się w plikach `ArrayInsertionTest.swift` oraz `ArrayInsertionTest.m`, a wyniki przedstawione zostały na wykresie 4.1. Z wykresu tego wyraźnie widać, że w obu językach dodawanie elementów do tablicy jest operacją działającą w czasie asymptotycznie liniowym, w języku Swift jest on jednak o około połowę krótszy.

Tak jak większość popularnych języków, zarówno Objective-C, jak i Swift zapewniają ciągłość pamięci tablicy. Z wykresu wynika, że oba języki rezerwują za wczasu większą ilość pamięci niż jest rzeczywiście potrzebna, a w razie przekroczenia już zajętej pamięci zwiększają rozmiar eksponencjalnie (na wykresie widać trzy takie miejsca, przy pamięci na 800000, 1600000 oraz 3200000 elementów).

4.2.1. Analiza działania

Rysunek 4.2 przedstawia wyniki profilowania w programie Instruments testu `ArrayInsertion` dla $n = 500000000$. Zgodnie z wykresem 4.1, czas działania testu w języku Objective-C jest około dwukrotnie dłuższy niż takiego samego testu w

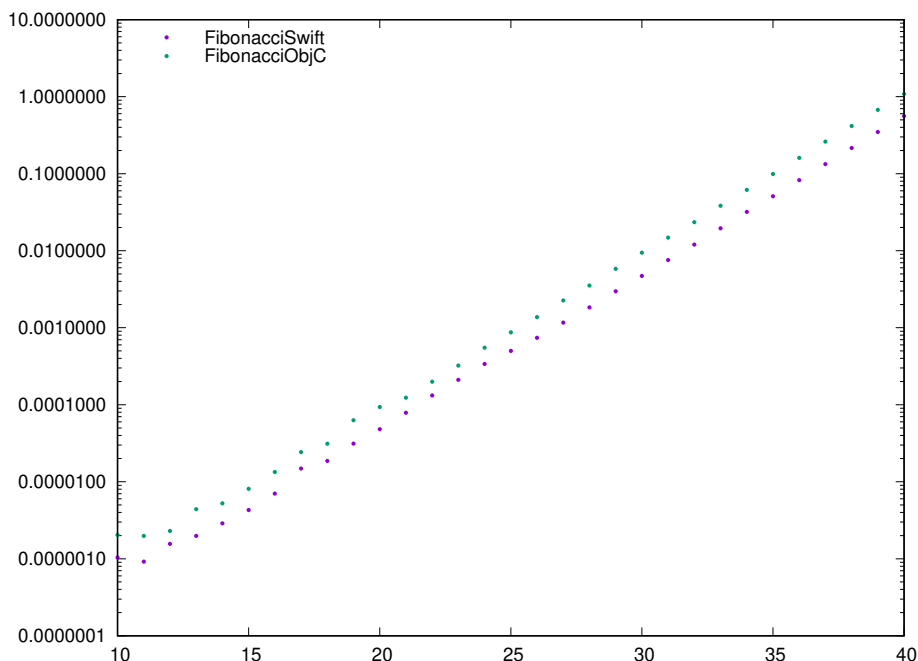
| Weight▼ | Self Weight | Symbol Name |
|----------------|-------------|--|
| 6.96 s 100.0% | 180.00 ms | ▼ArrayInsertionTestSwift.run() Benchmarks |
| 4.82 s 69.2% | 0 s | ▼specialized Array_copyToNewBuffer(oldCount:) Benchmarks |
| 4.82 s 69.2% | 0 s | ▼specialized _ArrayBufferProtocol_arrayOutOfPlaceUpdate<A>(_::_:_:) Benchmarks |
| 4.36 s 62.6% | 4.36 s | _platform_memmove\$VARIANT\$Haswell libsystem_platform.dylib |
| 461.00 ms 6.6% | 0 s | ▼_swift_release_dealloc Benchmarks |
| 461.00 ms 6.6% | 0 s | ▼free_large libsystem_malloc.dylib |
| 443.00 ms 6.3% | 0 s | ▼mvm_deallocate_pages libsystem_malloc.dylib |
| 443.00 ms 6.3% | 0 s | ▼mach_vm_deallocate libsystem_kernel.dylib |
| 443.00 ms 6.3% | 443.00 ms | _kernelrpc_mach_vm_deallocate_trap libsystem_kernel.dylib |
| 18.00 ms 0.2% | 18.00 ms | madvise libsystem_kernel.dylib |
| 1.35 s 19.3% | 1.35 s | specialized Array_appendElementAssumeUniqueAndCapacity(_:newElement:) Benchmarks |
| 254.00 ms 3.6% | 0 s | ▼_swift_release_dealloc Benchmarks |
| 254.00 ms 3.6% | 0 s | ▼free_large libsystem_malloc.dylib |
| 254.00 ms 3.6% | 0 s | ▼mvm_deallocate_pages libsystem_malloc.dylib |
| 254.00 ms 3.6% | 0 s | ▼mach_vm_deallocate libsystem_kernel.dylib |
| 254.00 ms 3.6% | 254.00 ms | _kernelrpc_mach_vm_deallocate_trap libsystem_kernel.dylib |
| 198.00 ms 2.8% | 0 s | ▼specialized protocol witness for _IndexableBase.formIndex(after:) in conformance <A> CountableRange<A> Benchmarks |
| 198.00 ms 2.8% | 0 s | ▼specialized _Indexable.formIndex(after:) Benchmarks |
| 198.00 ms 2.8% | 0 s | ▼specialized protocol witness for _IndexableBase.index(after:) in conformance <A> CountableRange<A> Benchmarks |
| 198.00 ms 2.8% | 0 s | ▼specialized CountableRange.index(after:) Benchmarks |
| 198.00 ms 2.8% | 198.00 ms | protocol witness for _Strideable.advanced(by:) in conformance Int Benchmarks |
| 161.00 ms 2.3% | 161.00 ms | specialized Array_getCount() Benchmarks |

| Weight▼ | Self Weight | Symbol Name |
|----------------|-------------|--|
| 11.25 s 100.0% | 822.00 ms | ▼-[ArrayInsertionTestObjC run] Benchmarks |
| 7.71 s 68.5% | 2.55 s | ▼-[__NSArrayM insertObject:atIndex:] CoreFoundation |
| 3.07 s 27.2% | 3.07 s | _platform_memmove\$VARIANT\$Haswell libsystem_platform.dylib |
| 1.76 s 15.6% | 1.76 s | _platform_bzero\$VARIANT\$Haswell libsystem_platform.dylib |
| 335.00 ms 2.9% | 0 s | ▼free_large libsystem_malloc.dylib |
| 321.00 ms 2.8% | 0 s | ▼mvm_deallocate_pages libsystem_malloc.dylib |
| 321.00 ms 2.8% | 0 s | ▼mach_vm_deallocate libsystem_kernel.dylib |
| 321.00 ms 2.8% | 321.00 ms | _kernelrpc_mach_vm_deallocate_trap libsystem_kernel.dylib |
| 14.00 ms 0.1% | 14.00 ms | madvise libsystem_kernel.dylib |
| 762.00 ms 6.7% | 762.00 ms | -[__NSArrayM addObject:] CoreFoundation |
| 668.00 ms 5.9% | 668.00 ms | +([NSNumber numberWithInt:] Foundation |
| 443.00 ms 3.9% | 443.00 ms | objc_release libobjc.A.dylib |
| 316.00 ms 2.8% | 316.00 ms | objc_retainAutoreleasedReturnValue libobjc.A.dylib |
| 300.00 ms 2.6% | 300.00 ms | objc_retain libobjc.A.dylib |
| 226.00 ms 2.0% | 226.00 ms | -[ArrayInsertionTestObjC numberOfInsertions] Benchmarks |

Rysunek 4.2: Tablica profilowania dla testu ArrayInsertion (u góry - Swift, na dole - Objective-C)

Swift. Tablica profilowania wskazuje potencjalne powody takiego stanu:

- Kod funkcji `run` w teście języka Objective-C składa się z utworzenia modyfikowalnej tablicy (operacja ta ma znikomy czas, nie została wylistowana na tablicy profilowania) oraz zwykłej pętli `for` przebiegającej od 0 do `numberOfInsertions`. W związku z tym można założyć, że czas działania pętli `for` to czas działania samej funkcji (oznaczony na tablicy profilowania jako *Self Weight*) plus czas dostępu do właściwości `numberOfInsertions` (kolor fioletowy na tablicy). Sumarycznie czas ten wynosi 1,05s. Kod testu dla języka Swift jest bardzo podobny, jedyną różnicą jest zastąpienie pętli `for` pętlą `for...in`, która w swojej implementacji używa protokołu `Indexable` i jego odpowiednich implementacji. Samo użycie tego protokołu wymaga 359 ms (instrukcje oznaczone kolorem fioletowym na tablicy profilowania), jednak czas działania samej funkcji `run` to zaledwie 180 ms, co daje łączny czas 539ms, czyli prawie o połowę krótszy niż w przypadku języka Objective-C. Zatem mimo, że pętla w języku Swift wydają się być bardziej skomplikowane, są wydajniejsze niż proste pętla `for` z Objective-C.
- W Objective-C kod funkcji `addObject` został `inline`'owany do wywołań metod `insertObjectAtIndex` oraz `addObject` klas wewnętrznych. Łączny czas wywołań tych dwóch metod to 3,31s. W Swift metoda `append` również została `inline`'owana, tym razem do metod prywatnych `_copyToNewBuffer(oldCount:)` oraz `_appendElementAssumeUniqueAndCapacity(_:newElement:)`, łączny czas wywołania tych metod to jedynie 1,35 s. Można więc wyciągnąć wniosek, że sama implementacja tych metod jest mniej wydajna w Objective-C.
- Przenoszenie pamięci (kolor zielony na rysunku) trwa podobny czas w obu językach (4,36s dla Swift i 4,82 dla Objective-C). Jest to dosyć oczywiste, że względu na fakt, że operacje na pamięci są wykonywane przez systemową bibliotekę `system_platform`, zatem w dla obu języków została użyta ta sama implementacja, stąd czas jest podobny.
- Dealokowanie zajętej już pamięci (kolor czerwony na rysunku) zajmuje znacząco więcej czasu w Swift (715 ms) niż w Objective-C (335 ms). Powodem jednak nie jest sama implementacja procedury dealokowania, gdyż ta znajduje się w bibliotece `system_malloc` i jest taka sama w obu językach, ale samo wykorzystanie mechanizmu dealokacji w obu językach.
- Jak wspomniano w rozdziale 3.1.2., ARC jest rozszerzeniem języka dodanym kilkanaście lat po premierze samego Objective-C, przez co w tablicy profilowania widnieją metody przeznaczone do tego zadania (kolor żółty na rysunku). Jak widać, ich wywołania zajmują około 10% czasu wykonywania całego programu (1,05s)



Rysunek 4.3: Wykres czasu działania testu FibonacciTest dla obu języków

- Tablice w Objective-C mogą zawierać tylko obiekty dziedziczące po klasie `NSObject`. Typ całkowitoliczbowy, jest typem prostym, należy go zatem schować w obiekcie klasy `NSNumber`, który można dodać do tablicy. Taka operacja to jednak dodatkowe 668ms do czasu działania programu (kolor niebieski).

4.3. Rekurencyjne obliczanie liczby Fibonacciego

Test polega na obliczeniu n -tej liczby Fibonacciego za pomocą naiwnego algorytmu rekurencyjnego. Kod obu testów znajduje się w plikach `FibonacciTest.m` oraz `FibonacciTest.swift`. Wyniki obu testów dla wartości $n \in < 5, 40 >$ zostały zwizualizowane na wykresie 4.3. Czas działania obu programów rośnie oczywiście wykładniczo, z wykresu wynika jednak, że program napisany w Swift jest około 40% szybszy niż jego odpowiednik napisany w języku Objective-C.

4.3.1. Analiza działania

Tablica profilowania 4.4 obu programów pozwala znaleźć powody różnicy prędkości pomiędzy dwoma językami. Główną różnicą jest wywołanie funkcji `objc_msgSend` w teście napisanym w Objective-C. Wywołania te są odpowiedzialne za ponad 5 sekund czasu działania testu, co daje prawie 37% całego czasu działania testu. Więcej o wywoływaniu metod w Objective-C napisano w sekcji 3.2.3.. Przykład ten dobrze obrazuje problemy z wydajnością języka Objective-C przy bardzo dużej ilości

| Weight▼ | Self Weight | Symbol Name |
|----------------|-------------|--|
| 7.05 s 100.0% | 0 s | ▼FibonacciTestSwift.run() Benchmarks |
| 7.05 s 100.0% | 6.84 s | ▼FibonacciTestSwift.fibonacci(n:) Benchmarks |
| 208.00 ms 2.9% | 0 s | ▼<Unknown Address> |
| 208.00 ms 2.9% | 208.00 ms | FibonacciTestSwift.fibonacci(n:) Benchmarks |

| Weight▼ | Self Weight | Symbol Name |
|----------------|-------------|---|
| 13.98 s 100.0% | 8.62 s | ▼-[FibonacciTestObjC fibonacciForN:] Benchmarks |
| 5.15 s 36.8% | 5.15 s | objc_msgSend libobjc.A.dylib |
| 204.00 ms 1.4% | 0 s | ▼<Unknown Address> |
| 204.00 ms 1.4% | 204.00 ms | -[FibonacciTestObjC fibonacciForN:] Benchmarks |

Rysunek 4.4: Tablica profilowania dla testu ArrayInsertion (u góry - Swift, na dole - Objective-C)

wywołań metod klas.

Drugim powodem, dla którego kod napisany w Objective-C jest wolniejszy jest samo działanie ciała metody `[fibonacciForN:]`, które jest o ponad 2 sekundy wolniejsze od analogicznego kodu napisanego w Swift.

4.4. Sortowanie bąbelkowe

Sortowanie bąbelkowe to jeden z najprostszych algorytmów sortowania. Jego największą zaletą jest bardzo prosta i intuicyjna implementacja. Test opisany w tym rozdziale polega na zaimplementowaniu algorytmu sortowania bąbelkowego i uruchomieniu go na tablicy o wielkości $n \in < 1000, 15000 >$ wypełnionej losowymi 32-bitowymi liczbami całkowitymi.

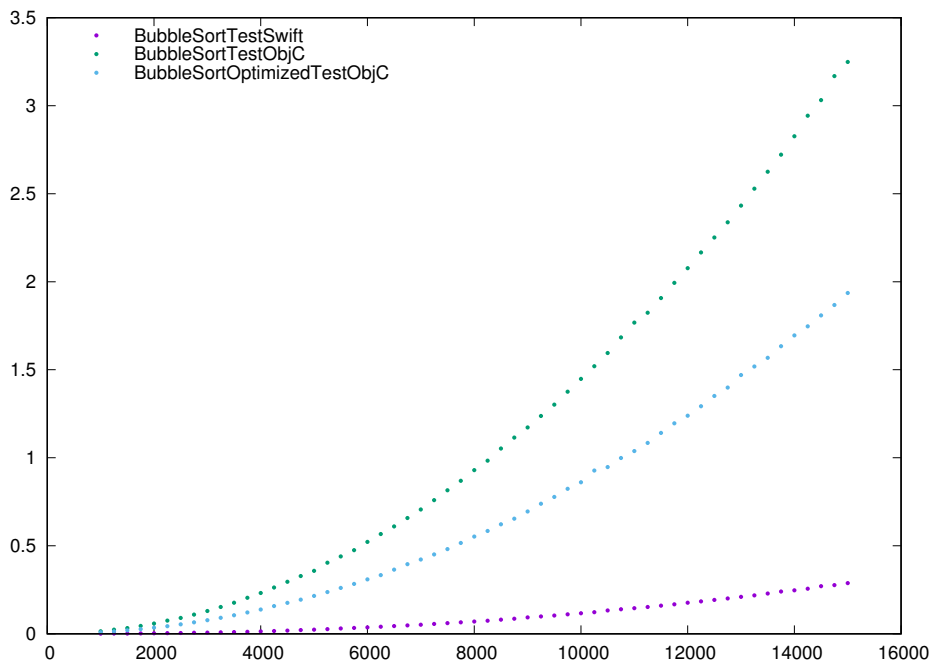
4.4.1. Analiza działania

Jak wynika z wykresu, implementacja w Swift jest ponad 10-krotnie szybsza niż najprostsza implementacja w Objective-C. Posortowanie tablicy 15000 elementów zajmuje tylko $288ms$ w przypadku kodu w Swift i $3,2s$ dla Objective-C.

Rysunek 4.6 przedstawia tablicę profilowania testu `BubbleSort` zaimplementowanego w języku Swifta (na górze) oraz w Objective-C (po środku) dla tablicy z 30000 elementów.

Tablica profilowania testu napisanego w języku Swift jest bardzo krótka - aż 93% czasu wykonywania testu zajęły dwie operacje:

- około $0,5s$ zajęło porównywanie elementów w tablicy (oznaczone kolorem czerwonym)



Rysunek 4.5: Wykres czasu działania testu BubbleSortTest dla obu języków

- wywołanie operacji zdefiniowanych w samej metodzie `BubbleSort(:_)` zajęło około $800ms$. Te operacje to obsługa pętli `for` operacje arytmetyczne związane z przesuwaniem indeksów oraz zamiana elementów w tablicy.

Analiza implementacji w Objective-C jest nieco bardziej skomplikowana. Najwięcej czasu zajęło wysyłanie wiadomości do obiektów za pomocą metody `objc_msgSend` (oznaczonej kolorem żółtym na tablicy profilowania). Czas wszystkich wywołań tej metody wyniósł $4,58s$.

Drugą najbardziej czasochłonną grupą operacji były operacje odczytu i zapisu do/z tablicy (kolorem zielonym na tablicy profilowania). Metody `objectAtIndexedSubscript:` i `setObject:atIndexedSubscript` są wywoływane po użyciu operatora indeksowania na tablicy `NSArray` i łącznie czas ich wywołania zajmuje $4,31s$.

Wykonanie operacji z ciała metody `run` (kolor czerwony), trwało łącznie $2,5s$. Operacje te to głównie obliczenia arytmetyczne oraz obsługa pętli `for`. Wywołania funkcji oznaczone kolorem niebieskim na tablicy profilowania dotyczą zarządzania licznikami referencji i łącznie również prawie $2,5s$.

Ciekawym przypadkiem pokazującym kosztowność systemu wysyłania wiadomości jest sprawdzanie wartości zapisanej do właściwości `n`. W Objective-C dla każdej właściwości o nazwie `x` kompilator generuje:

- zmienną o nazwie `_x`
- getter o nazwie `x`

| Weight▼ | Self Weight | Symbol Name |
|-----------------|-------------|--|
| 1.39 s 100.0% | 0 s | ▼BubbleSortTestSwift.run() Benchmarks |
| 1.39 s 99.9% | 0 s | ▼BubbleSortTestSwift.bubbleSort(,:) Benchmarks |
| 1.39 s 99.9% | 801.00 ms | ▼specialized BubbleSortTestSwift.bubbleSort(,:) Benchmarks |
| 499.00 ms 35.8% | 499.00 ms | protocol witness for static Equatable,== infix(.,:) in conformance Int Benchmarks |
| 92.00 ms 6.6% | 92.00 ms | specialized Array._getElement(_:wasNativeTypeChecked:matchingSubscriptCheck:) Benchmarks |
| 1.00 ms 0.0% | 1.00 ms | os_unfair_lock_unlock libsystem_platform.dylib |

| Weight▼ | Self Weight | Symbol Name |
|----------------|-------------|--|
| 14.20 s 100.0% | 0 s | ▼-[BubbleSortTestObjC run] Benchmarks |
| 9.42 s 66.3% | 2.53 s | ▼-[BubbleSortTestObjC bubbleSortWithArray:] Benchmarks |
| 2.51 s 17.6% | 2.51 s | -[_NSArrayM objectAtIndexedSubscript:] CoreFoundation |
| 1.80 s 12.6% | 1.80 s | -[_NSArrayM setObject:atIndexedSubscript:] CoreFoundation |
| 837.00 ms 5.8% | 837.00 ms | objc_retain libobjc.A.dylib |
| 808.00 ms 5.6% | 808.00 ms | objc_release libobjc.A.dylib |
| 766.00 ms 5.3% | 766.00 ms | objc_retainAutoreleasedReturnValue libobjc.A.dylib |
| 172.00 ms 1.2% | 172.00 ms | ▼-[BubbleSortTestObjC n] Benchmarks |
| 4.58 s 32.2% | 4.58 s | objc_msgSend libobjc.A.dylib |
| 151.00 ms 1.0% | 151.00 ms | DYLD-STUB\$\$objc_retainAutoreleasedReturnValue Benchmarks |
| 43.00 ms 0.3% | 0 s | ▼<Unknown Address> |
| 30.00 ms 0.2% | 30.00 ms | -[_NSArrayM objectAtIndexedSubscript:] CoreFoundation |
| 6.00 ms 0.0% | 6.00 ms | objc_retain libobjc.A.dylib |
| 5.00 ms 0.0% | 5.00 ms | objc_release libobjc.A.dylib |
| 1.00 ms 0.0% | 1.00 ms | ▼-[BubbleSortTestObjC n] Benchmarks |
| 1.00 ms 0.0% | 1.00 ms | -[_NSArrayM setObject:atIndexedSubscript:] CoreFoundation |
| 1.00 ms 0.0% | 1.00 ms | arc4random libsystem_c.dylib |
| 1.00 ms 0.0% | 1.00 ms | +([NSNumber numberWithIntUnsignedInt:]) Foundation |

| Weight▼ | Self Weight | Symbol Name |
|----------------|-------------|--|
| 8.20 s 100.0% | 1.58 s | ▼-[BubbleSortOptimizedTestObjC run] Benchmarks |
| 1.83 s 22.3% | 1.83 s | objc_msgSend libobjc.A.dylib |
| 1.61 s 19.6% | 1.61 s | -[_NSArrayM objectAtIndexedSubscript:] CoreFoundation |
| 1.57 s 19.1% | 1.57 s | -[_NSArrayM exchangeObjectAtIndex:withObjectAtIndex:] CoreFoundation |
| 537.00 ms 6.5% | 537.00 ms | objc_retain libobjc.A.dylib |
| 528.00 ms 6.4% | 528.00 ms | objc_release libobjc.A.dylib |
| 523.00 ms 6.3% | 523.00 ms | objc_retainAutoreleasedReturnValue libobjc.A.dylib |
| 26.00 ms 0.3% | 0 s | ▼<Unknown Address> |
| 24.00 ms 0.2% | 24.00 ms | -[_NSArrayM exchangeObjectAtIndex:withObjectAtIndex:] CoreFoundation |
| 2.00 ms 0.0% | 2.00 ms | -[_NSArrayM objectAtIndexedSubscript:] CoreFoundation |
| 1.00 ms 0.0% | 0 s | ▼arc4random libsystem_c.dylib |
| 1.00 ms 0.0% | 1.00 ms | DYLD-STUB\$\$_platform_memset libsystem_platform.dylib |

Rysunek 4.6: Tablica profilowania dla testu BubbleSort (u góry - Swift, po środku - Objective-C, na dole - Objective-C po optymalizacjach)

- setter `setX`:

Z tego powodu, odczytanie wartości z właściwości `n` jest tak naprawdę wywołaniem metody `n` i odczytanie wartości zmiennej ukrytej za właściwością. Z punktu widzenia kompilatora, odczyt taki jest zatem niczym innym niż wywołaniem metody, czyli wysłaniem wiadomości, stąd też samo odczytanie zmiennej zajmuje *172ms* (kolor szary na tablicy profilowania).

Z powyższej analizy można łatwo wyznaczyć fragmenty kodu, które powinny zostać zoptymalizowane:

- wywoływanie operatora indeksowania - zamiast zamieniać miejscami elementy za pomocą czterech wywołań operatora indeksowania, można użyć metody `exchangeObjectAtIndex:withObjectAtIndex:` klasy `NSArray`
- wysyłanie wiadomości do obiektów za pomocą `objc_msgSend` - metoda `exchangeObjectAtIndex:withObjectAtIndex:` jest za każdym razem wywoływana na tym samym obiekcie, dlatego też można już wcześniej wziąć wskaźnik na tą metodę (w nomenklaturze Objective-C - selektor), zachować referencję do niego i bezpośrednio wywoływać
- użycie właściwości `n` - zamiast przy każdym sprawdzeniu warunku pętli odwoływać się do właściwości `n`, można ją zachować na stosie jako zmienną lokalną funkcji i odwoływać się do niej bezpośrednio

Klasa `BubbleSortOptimizedTestObjC` zawiera implementację wyżej wymienionych optymalizacji, wyniki działania zostały zawarte również na wykresie 4.5 oraz tablicy profilowania 4.6. Jak widać, zaproponowane optymalizacje znacznie przyspieszyły działanie algorytmu o około 40% względem kodu bez optymalizacji.

Największy przyrost wydajności zaobserwowano dla wywołań funkcji `objc_msgSend`, czas 4,58s udało się zmniejszyć do 1,83s (spadek o 60%). Główne powody to zastąpienie kilku wywołań operatorów indeksowania jednym wywołaniem metody zamieniającej elementy miejscami oraz trzymanie referencji na tą metodę w zmiennej lokalnej. Spadł również czas wywołania metod związanych z operatorem indeksowania - pozbyto się zupełnie użycia metody ustawiającej wartość, a czas zużywany na metodę pobierającą wartość spadł z 2,51s do 1,61. Łącznie czas operacji na tablicy spadł z 4,31s do 3,18s (spadek o 25%). Czas obsługi liczników referencji spadł z 2,4s do 1,6s, co oznacza spadek o około 35%, a czas poświęcony wcześniej na odczyt właściwości `n` spadła do wartości niezauważalnej podczas profilowania.

Pomimo tak znacznego przyspieszenia, kod ten nadal jest około 6-krotnie wolniejszy od implementacji w Swift.

4.5. Budowanie binarnego drzewa poszukiwań

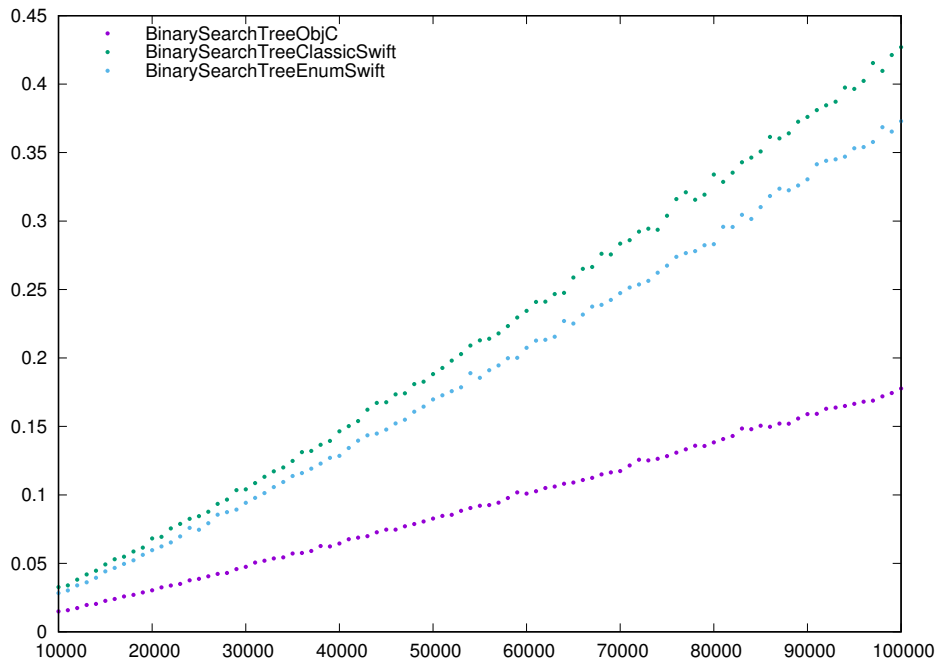
Test opisany w tym rozdziale polega na utworzeniu binarnego drzewa poszukiwań o n elementach. Implementacja w języku Objective-C jest klasyczną implementacją tego problemu. Zdefiniowana została struktura `Node` przechowująca wskaźniki na lewe i prawe pod-węzeł oraz daną wartość.

Nieco ciekawiej wygląda kwestia implementacji algorytmu budowania drzewa w języku Swift. Po pierwsze, programiści Swift często przedkładają bezpieczeństwo i niezawodność kodu nad jego wydajność. Przykładowo, zby zapewnić bezpieczeństwo dostępu do danych (np. podczas dostępu z wielu wątków) stosuje się struktury niemutowalne. Po drugie, Swift oferuje dużo więcej narzędzi niż Objective-C, m.in. funkcje wyższego rzędu jako (***TODO: Przetłumaczyć***) ang. *first class citizens* czy algebraiczne typy danych w postaci typów wyliczeniowych z wartościami powiązanymi. Z tych dwóch powodów, poniższy test zawiera dwie implementacje w języku Swift. Obie zapewniają niemutowalność bieżących elementów drzewa - dodanie nowego elementu powoduje utworzenie nowego węzła, a nie zmodyfikowanie już istniejącego. Pierwsza implementacja to implementacja "klasyczna". Dane przechowywane są w klasie `Tree` (odpowiednik klasy `Node` z Objective-C), a metoda dodająca element przechodzi po drzewie i tworzy nowy obiekt typu `Tree` w odpowiednim miejscu. Druga implementacja jest oparta na wykorzystaniu typów wyliczeniowych z wartościami powiązanymi i wygląda nieco bardziej jak kod znany z języków funkcyjnych. Dane przechowywane są tutaj w typie wyliczeniowym `Tree`, który definiuje dwa "konstruktory typów": `empty` oraz `node(Tree, Int, Tree)`. Funkcja dodająca element używa pattern matchingu, aby znaleźć odpowiednie miejsce w drzewie i tam dodaje nowy obiekt. Kod opisanych testów znajduje się odpowiednio w plikach `BinarySearchTreeTest.m`, `BinarySearchTreeClassicTest.swift` oraz `BinarySearchTreeEnumsTest.swift`.

4.5.1. Analiza działania

Wykres 4.7 Przedstawia wyniki działania testu tworzącego drzewo binarne o wielkości n . Jak widać, jest to pierwszy test, którym implementacja w Objective-C jest szybsza od implementacji w Swift. Kod napisany w starszym z języków wykonuje się około 2 razy szybciej od implementacji napisanej Swift przy pomocy typów wyliczeniowych i ponad dwa razy szybciej od "klasycznej" implementacji w Swift.

Rysunek 4.8 przedstawia tabelę profilowania ww. testów dla wejściowego rozmiaru drzewa $n = 5000000$. Dla czytelności pominięte zostały wpisy, których czas wykonywania nie przekraczał 0.1% łącznego czasu wykonywania testu. Podczas tego pojedynczego uruchomienia kod napisany w Objective-C okazał się być aż 2,5 raza szybszy od implementacji w Swift przy użyciu typów wyliczeniowych oraz 3 razy szybszy od klasycznej implementacji w Swift. Głównym powodem takiego stanu rzeczy była wspomniana we wstępie do tego testu niemutowalność struktury drzewa



Rysunek 4.7: Wykres czasu działania testu BinarySearchTree

dla implementacji w Swift.

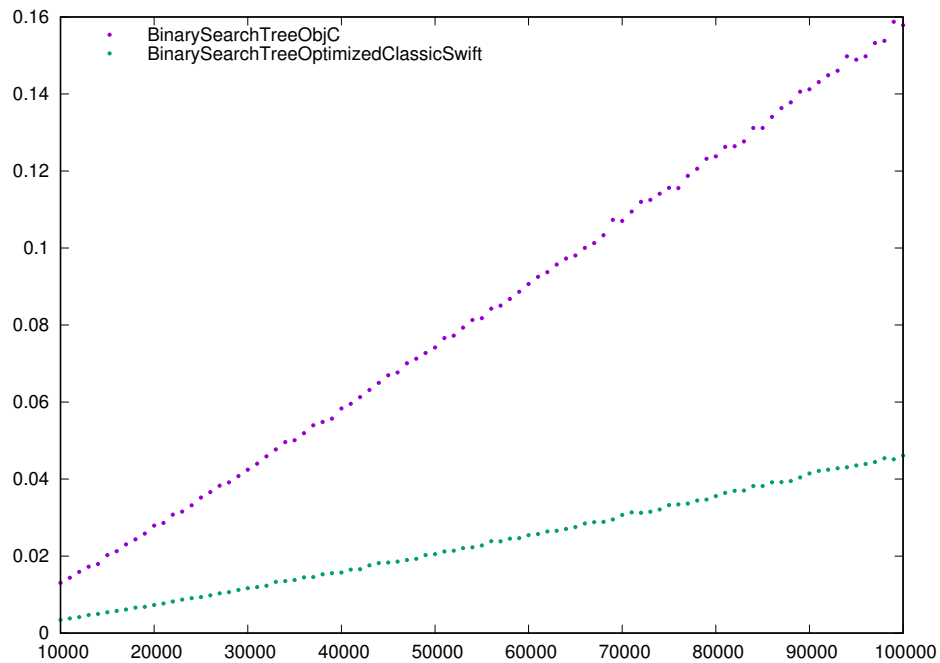
Największą ilość czasu podczas wykonywania testów w Swift zajęło tworzenie nowych obiektów (kolor niebieski na tablicy profilowania) oraz obsługa ARC (kolor czerwony). Powodem, dla którego tak się działo jest wielokrotne tworzenie obiektów dla tych samych węzłów, co powodowało bardzo częste wywoływania inicjalizatora i deinicjalizatora oraz w późniejszym etapie funkcji odpowiadających za inkrementację i dekrementację licznika referencji. Dla implementacji w Swifta z użyciem typów wyliczeniowych łączny czas wywołań inicjalizatorów i deinicjalizatorów wyniósł 25,04s, a operacje obsługi liczników referencji zajęły łącznie 13,33s. Dla kodu napisanego za pomocą klas w Swift wartości te wynosiły odpowiednio 32,63s dla inicjalizacji i deinicjalizacji oraz 12,41s dla obsługi ARC. W Objective-C czas tworzenia nowych obiektów wynosił zaledwie 445ms. Obsługa ARC zajęła łącznie ok 11,64s. Dodatkowo do czasu działania kodu Objective-C należy dodać standardowo czas przesyłania metod za pomocą funkcji `objc_msgSend`, który wyniósł 2,01s.

Z powyższej analizy wynika, że choć w Swift można napisać bardzo bezpieczny kod, to ma to jednak swoją cenę w postaci mocno obniżonej wydajności. Analiza nie byłaby jednak pełna, jeśli nie zostałaby przeprowadzona próba napisania kod działającego identycznie jak kod Objective-C, tj. modyfikująca aktualne węzły w drzewie.

Wykres 4.9 przedstawia czas działania zoptymalizowanego testu napisanego w języku Swift. Jak widać, test ten jest teraz około 3-krotnie szybszy od testu zaimplementowanego w Objective-C. Tablica profilowania 4.10 tego testu wskazuje wyraź-

| | | | |
|-----------|--------|-----------|--|
| 15.58 s | 100.0% | 23.00 ms | ▼-[BinarySearchTreeTestObjC run] Benchmarks |
| 15.24 s | 97.7% | 27.00 ms | ▼-[BinarySearchTreeObjC addElement:] Benchmarks |
| 14.94 s | 95.8% | 594.00 ms | ▼-[BinarySearchTreeObjC addElement:toTreeRootedInNode:] Benchmarks |
| 6.89 s | 44.2% | 6.89 s | objc_retain libobjc.A.dylib |
| 4.30 s | 27.6% | 4.30 s | objc_release libobjc.A.dylib |
| 1.91 s | 12.2% | 1.91 s | objc_msgSend libobjc.A.dylib |
| 445.00 ms | 2.8% | 10.00 ms | ▶+[NSObject new] libobjc.A.dylib |
| 195.00 ms | 1.2% | 0 s | ▶<Unknown Address> |
| 182.00 ms | 1.1% | 182.00 ms | objc_retainAutoreleasedReturnValue libobjc.A.dylib |
| 159.00 ms | 1.0% | 159.00 ms | -[Node element] Benchmarks |
| 88.00 ms | 0.5% | 88.00 ms | -[Node left] Benchmarks |
| 87.00 ms | 0.5% | 87.00 ms | DYLD-STUB\$\$objc_retainAutoreleasedReturnValue Benchmarks |
| 44.00 ms | 0.2% | 9.00 ms | ▶objc_storeStrong libobjc.A.dylib |
| 90.00 ms | 0.5% | 90.00 ms | objc_retain libobjc.A.dylib |
| 88.00 ms | 0.5% | 88.00 ms | objc_release libobjc.A.dylib |
| 72.00 ms | 0.4% | 72.00 ms | objc_msgSend libobjc.A.dylib |
| 197.00 ms | 1.2% | 10.00 ms | ▶+[NSArray(generate) generateArrayOfSize:] Benchmarks |
| 41.00 ms | 0.2% | 41.00 ms | objc_release libobjc.A.dylib |
| 34.00 ms | 0.2% | 8.00 ms | ▶-[NSCFNumber longValue] CoreFoundation |
| 31.00 ms | 0.1% | 31.00 ms | objc_msgSend libobjc.A.dylib |
| 47.08 s | 100.0% | 0 s | ▼@objc BinarySearchTreeClassicTestSwift.run() Benchmarks |
| 46.91 s | 99.6% | 0 s | ▼static Tree.buildTree(elements:) Benchmarks |
| 46.84 s | 99.4% | 2.00 ms | ▼specialized static Tree.buildTree(elements:) Benchmarks |
| 25.74 s | 54.6% | 0 s | ▼think for @callee_guaranteed (@owned Tree, @unowned Int) -> (@owned Tree, @error @owned Error) Benchmarks |
| 25.74 s | 54.6% | 1.00 ms | ▼closure #1 in static Tree.buildTree(elements:) Benchmarks |
| 25.73 s | 54.6% | 764.00 ms | ▼Tree.add(element:) Benchmarks |
| 12.18 s | 25.8% | 12.18 s | _swift_retain_ Benchmarks |
| 11.62 s | 24.6% | 27.00 ms | ▶Tree.__allocating_init(left:value:right:) Benchmarks |
| 1.00 s | 2.1% | 1.00 s | _swift_release_ Benchmarks |
| 68.00 ms | 0.1% | 68.00 ms | swift_rt_swift_allocObject Benchmarks |
| 50.00 ms | 0.1% | 50.00 ms | swift_rt_swift_release Benchmarks |
| 48.00 ms | 0.1% | 48.00 ms | swift_rt_swift_retain Benchmarks |
| 21.01 s | 44.6% | 4.00 ms | ▶_swift_release_dealloc Benchmarks |
| 58.00 ms | 0.1% | 58.00 ms | _swift_release_n_ Benchmarks |
| 164.00 ms | 0.3% | 6.00 ms | ▶static Array<A>.generate(size:) Benchmarks |
| 39.55 s | 100.0% | 0 s | ▼@objc BinarySearchTreeEnumsTestSwift.run() Benchmarks |
| 39.39 s | 99.5% | 0 s | ▼static Tree.buildTree(elements:) Benchmarks |
| 39.34 s | 99.4% | 0 s | ▼specialized static Tree.buildTree(elements:) Benchmarks |
| 25.03 s | 63.2% | 0 s | ▼think for @callee_guaranteed (@owned Tree, @unowned Int) -> (@owned Tree, @error @owned Error) Benchmarks |
| 25.03 s | 63.2% | 2.00 ms | ▼closure #1 in static Tree.buildTree(elements:) Benchmarks |
| 25.00 s | 63.1% | 854.00 ms | ▼Tree.add(element:) Benchmarks |
| 10.77 s | 27.2% | 348.00 ms | ▶_swift_allocObject_ Benchmarks |
| 8.98 s | 22.7% | 8.98 s | _swift_retain_ Benchmarks |
| 2.68 s | 6.7% | 2.68 s | _swift_release_ Benchmarks |
| 1.43 s | 3.6% | 1.43 s | _swift_retain_n_ Benchmarks |
| 102.00 ms | 0.2% | 102.00 ms | swift_rt_swift_retain Benchmarks |
| 90.00 ms | 0.2% | 90.00 ms | swift_rt_swift_release Benchmarks |
| 40.00 ms | 0.1% | 40.00 ms | swift_rt_swift_allocObject Benchmarks |
| 14.23 s | 35.9% | 10.00 ms | ▶_swift_release_dealloc Benchmarks |
| 45.00 ms | 0.1% | 45.00 ms | _swift_release_n_ Benchmarks |

Rysunek 4.8: Tablica profilowania dla testu BinarySearchTree (u góry - Objective-C, po środku - Swift przy użyciu klas, na dole - Swift przy użyciu typów wyliczeniowych)



Rysunek 4.9: Wykres czasu zoptymalizowanej wersji w Swift dla BinarySearchTree w porównaniu z wersją w Objective-C

| | | | |
|-----------|--------|-----------|--|
| 8.75 s | 100.0% | 0 s | ▼@objc BinarySearchTreeOptimizedClassicTestSwift.run() Benchmarks |
| 8.56 s | 97.8% | 0 s | ▼static Tree.buildTree(elements:) Benchmarks |
| 8.55 s | 97.7% | 0 s | ▼specialized static Tree.buildTree(elements:) Benchmarks |
| 8.44 s | 96.5% | 5.00 ms | ▼think for @callee_guaranteed (@unowned Int) -> (@error @owned Error) Benchmarks |
| 8.38 s | 95.8% | 445.00 ms | ▼Tree.add(element:) Benchmarks |
| 6.12 s | 69.9% | 6.12 s | _swift_retain_ Benchmarks |
| 1.09 s | 12.4% | 1.09 s | _swift_release_ Benchmarks |
| 611.00 ms | 6.9% | 2.00 ms | ▼Tree.__allocating_init(left:value:right:) Benchmarks |
| 431.00 ms | 4.9% | 84.00 ms | ►_swift_allocObject_ Benchmarks |
| 175.00 ms | 2.0% | 175.00 ms | Tree.init(left:value:right:) Benchmarks |
| 52.00 ms | 0.5% | 52.00 ms | swift_rt_swift_release Benchmarks |
| 50.00 ms | 0.5% | 50.00 ms | swift_rt_swift_retain Benchmarks |
| 52.00 ms | 0.5% | 52.00 ms | _swift_release_ Benchmarks |
| 56.00 ms | 0.6% | 56.00 ms | _swift_retain_ Benchmarks |
| 45.00 ms | 0.5% | 45.00 ms | _swift_release_ Benchmarks |
| 9.00 ms | 0.1% | 0 s | ▼specialized protocol witness for Collection.formIndex(after:) in conformance [A] Benchmarks |
| 9.00 ms | 0.1% | 9.00 ms | specialized Array.formIndex(after:) Benchmarks |
| 178.00 ms | 2.0% | 11.00 ms | ►static Array<A>.generate(size:) Benchmarks |

Rysunek 4.10: Tablica profilowania dla zoptymalizowanej wersji testu BinarySearchTree

nie na przyczynę poprawy wydajności. W zoptymalizowanej implementacji pozbyto się tworzenia dużej ilości nowych obiektów podczas dodawania nowego elementu. Wywołania inicjalizatora zajmują teraz jedynie 611ms, co oznacza co oznacza skrócenie czasu potrzebnego na inicjalizację obiektów o 98% (wcześniej - 32,63s). Czas obsługi ARC również uległ zmniejszeniu - z 11,64s do 7,47s (spadek o 35%).

Zatem pomimo początkowych problemów z wydajnością również i w tym teście kod napisany w Swift okazał się szybszy. Należy jednak uważać, z jakich właściwości języka się korzysta. Pisanie kodu w oparciu o idee zaczerpnięte z programowania funkcyjnego może nie być dobrym wyborem pod względem wydajności kodu.

4.6. Rodzaje wywołania metod

Język Swift obsługuje dwa rodzaje wywołań metod: statyczną (ang. *direct dispatch*) i dynamiczną (ang. *table dispatch*). Oprócz tego, ze względu na możliwość użycia kodu Swift w Objective-C, programista może również używać wywoływania poprzez wiadomości (ang. *message dispatch*), które jest domyślnym sposobem wywoływania w starszym z języków.

Wywoływanie statyczne jest uznawane za najszybszy sposób wykonywania metod. Główną cechą jest pełna wiedza o tym, która metoda ma zostać wykonana już w trakcie procesu kompilowania. Z tego powodu, kompilator może stosować dużą ilość optymalizacji, takich jak bezpośrednie odwołania do adresu funkcji czy wywołanie funkcji "w linii" (ang. *inline*). Główną jej wadą jest natomiast bardzo ograniczone wsparcie dla polimorfizmu, przez co większość języków implementuje dodatkowo inny rodzaj wywoływania metod (np. tablice wirtualne w C++). W Swift wywoływanie statyczne jest używane dla struktur oraz rozszerzeń protokołów (ang. *protocol extensions*).

Wywołanie dynamiczne to mechanizm zaimplementowany w większości współczesnych obiektowych języków programowania. Jego działanie opiera się głównie o tablice wirtualne. W Swift istnieją dwa rodzaje tablic wirtualnych:

- *virtual dispatch tables* - tablice wirtualne tworzone dla klas i służące do wywoływania odpowiednich metod klasy. Dzięki nim, możliwe jest dziedziczenie i przeładowywanie metod w klasach dziedziczących.
- *witness tables* - tablice wirtualne tworzone dla każdego typu implementującego protokół. Dzięki nim możliwe jest użycie protokołów.

Zaletą takiego podejścia jest jego elastyczność, która umożliwia zaimplementowanie takich elementów języka, jak dziedziczenie czy protokoły. Wadą natomiast jest dodatkowy czas, potrzebny na wyszukanie adresu funkcji w tablicy wirtualnej.

Uniemożliwia też stosowanie niektórych metod optymalizacji kodu. W Swift wywołanie dynamiczne jest domyślnie używane dla klas oraz protokołów.

Wywołanie przez wiadomości to mechanizm odziedziczony z Objective-C i używany głównie w trakcie wywoływania kodu Objective-C. Jest uważany za najwolniejszy z trzech dostępnych sposobów wywoływania funkcji, wspiera jednak funkcje języka, takie jak selector czy KVO (ang. *Key Value Observers*) bez których prawie niemożliwe byłoby używanie klas i bibliotek napisanych w Objective-C z poziomu kodu swiftowego.

Test opisany w tym rozdziale polega na wykonaniu bardzo prostej funkcji mnożącej liczbę całkowitą przez 2 na kilka sposobów:

- jako metoda struktury - metoda struktury zostanie wywołana statycznie. Kod tego eksperymentu znajduje się w teście `StaticDispatchTest.swift`.
- jako metoda struktury implementującej protokół - ze względu na fakt, że struktura jest schowana za protokołem i kompilator nie ma sposobu na zidentyfikowanie dokładnego typu obiektu, metoda zostanie wywołana dynamicznie. Kod dla tego eksperymentu znajduje się w teście `DynamicDispatchTest.swift`.
- jako metoda klasy z Objective-C - funkcje napisane w Objective-C są wywoływane przez wysłanie wiadomości. Kod dla tego testu znajduje się w pliku `MessageDispatchTest.mm`.

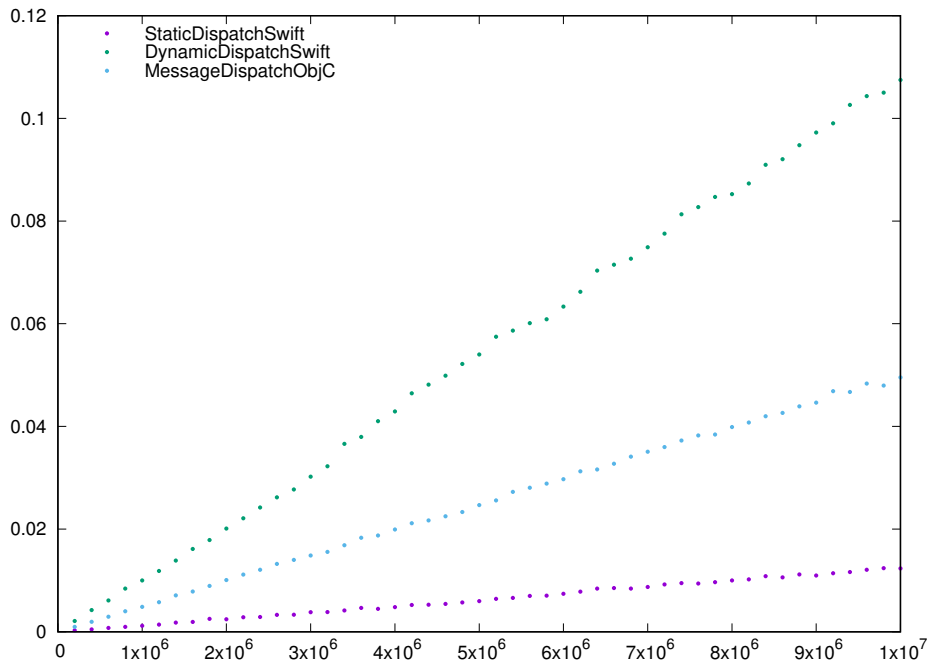
4.6.1. Analiza działania

Wyniki wyżej opisanych testów znajdują się na wykresie 4.11.

Jak widać na wykresie, czasy działania różnych metod wywołania różnią się od siebie znacznie. Zgodnie z przewidywaniami z opisu tego testu, statyczna metoda wywołania funkcji jest najszybsza. Na drugim miejscu znalazł się mechanizm wywoływania przez wiadomości, który jest ok. 4-krotnie wolniejszy od wywoływania statycznego. Wywołanie dynamiczne okazało się natomiast niespodziewanie wolne, zajęło ponad 2 razy więcej czasu niż wywołanie przez wiadomości i prawie 9 razy więcej od wywołania statycznego. Tablica profilowania z rysunku 4.12 wskazuje na pewne powody takiego stanu.

Tablica profilowania przedstawiona na rysunku 4.12 przedstawia czas działania testów dla różnych metod wywołania i parametru wejściowego $n = 20000000$. Na jej podstawie można wysnuć kilka wniosków odnośnie czasu działania testów.

Po pierwsze, na czas działania testu dla wywołania statycznego składa się jedynie obsługa pętli `forEach` (kolor żółty na tablicy profilowania) oraz wywołanie funkcji `multiply`. To właśnie brak dodatkowych kosztownych operacji powoduje, że wywołanie statyczne jest najszybszym z trzech porównywanych mechanizmów.



Rysunek 4.11: Wykres czasu działania trzech metod wywołania funkcji: statycznej, dynamicznej i przez wiadomości

Przy wywołaniu dynamicznym, implementacja funkcji `multiply` jest schowana za protokołem. Z tego powodu, kompilator musi w trakcie działania programu znaleźć adres do odpowiedniej implementacji, którą należy wywołać. Operacja ta zostanie wykonana w dwóch krokach:

- najpierw kompilator utworzy tablicę wirtualną ang. *Witness table* dla protokołu `MultiplyByTwoProtocol` oraz struktury `MultiplyByTwoStruct`. Tablica ta zawiera odnośniki do adresów funkcji dla typu implementującego dany protokół. W tym przypadku znajdzie się tam adres implementacji dla funkcji `multiply` w strukturze `MultiplyByTwoStruct`.
- następnie kompilator powiąże utworzoną tablicę z danym obiektem struktury tworząc tymczasowy typ egzystencyjny (ang. *existential type*). Taki obiekt zostanie dopiero przekazany do wywołania funkcji.

Z tablicy profilowania wynika, że tworzenie tablicy *Witness table* zajmuje 110ms, a tworzenie i usuwanie typu egzystencjalnego to czas 73ms, co łącznie daje aż 75% całego czasu wykonania funkcji (kolor czerwony na tablicy profilowania).

Wywołanie przez wiadomości okazało się szybsze niż wywołanie dynamiczne. Zawdzięcza to głównie świetnej optymalizacji oraz użyciu cache do przechowywania informacji o tym, jaka wiadomość powinna zostać wysłana. Jako, że wysyłana jest cały czas taka sama wiadomość do tej samej klasy, to cache jest tutaj bardzo

| | | | |
|-----------|--------|----------|---|
| 32.00 ms | 100.0% | 0 s | ▼@objc StaticDispatchSwift.run() Benchmarks |
| 20.00 ms | 62.5% | 0 s | ▼think for @callee_guaranteed (@unowned MultiplyByTwoStruct) -> (@error @owned Error) Benchmarks |
| 20.00 ms | 62.5% | 0 s | ▼closure #1 in StaticDispatchSwift.run() Benchmarks |
| 20.00 ms | 62.5% | 20.00 ms | MultiplyByTwoStruct.multiply() Benchmarks |
| 7.00 ms | 21.8% | 0 s | ▼specialized protocol witness for Collection.subscript.getter in conformance [A] Benchmarks |
| 7.00 ms | 21.8% | 0 s | ▼specialized Array.subscript.getter Benchmarks |
| 7.00 ms | 21.8% | 7.00 ms | specialized Array._getElement(_:wasNativeTypeChecked:matchingSubscriptCheck:) Benchmarks |
| 4.00 ms | 12.5% | 4.00 ms | protocol witness for static Equatable.== infix(.,:) in conformance Int Benchmarks |
| 1.00 ms | 3.1% | 0 s | ▼specialized protocol witness for Collection.formIndex(after:) in conformance [A] Benchmarks |
| 1.00 ms | 3.1% | 1.00 ms | specialized Array.formIndex(after:) Benchmarks |
| 242.00 ms | 100.0% | 0 s | ▼@objc DynamicDispatchSwift.run() Benchmarks |
| 66.00 ms | 27.2% | 56.00 ms | ▼outlined init with copy of MultiplyByTwoProtocol Benchmarks |
| 10.00 ms | 4.1% | 10.00 ms | swift::metadataimpl::BufferValueWitnesses<swift::metadataimpl::ValueWitnesses<swift::metadataimpl::NativeB... |
| 63.00 ms | 26.0% | 28.00 ms | ▼DynamicDispatchSwift.run() Benchmarks |
| 29.00 ms | 11.9% | 15.00 ms | ▼closure #1 in DynamicDispatchSwift.run() Benchmarks |
| 13.00 ms | 5.3% | 13.00 ms | protocol witness for MultiplyByTwoProtocol.multiply() in conformance MultiplyByTwoStruct Benchmarks |
| 1.00 ms | 0.4% | 1.00 ms | __swift_project_boxed_opaque_existential_1 Benchmarks |
| 4.00 ms | 1.6% | 4.00 ms | swift::metadataimpl::ValueWitnesses<swift::metadataimpl::NativeBox<unsigned long long, 8ul, 8ul, 8ul> >::dest |
| 1.00 ms | 0.4% | 1.00 ms | outlined init with take of MultiplyByTwoProtocol Benchmarks |
| 1.00 ms | 0.4% | 1.00 ms | outlined init with copy of MultiplyByTwoProtocol Benchmarks |
| 39.00 ms | 16.1% | 39.00 ms | __swift_project_boxed_opaque_existential_1 Benchmarks |
| 38.00 ms | 15.7% | 24.00 ms | ▼outlined init with take of MultiplyByTwoProtocol Benchmarks |
| 14.00 ms | 5.7% | 14.00 ms | swift::metadataimpl::BufferValueWitnesses<swift::metadataimpl::ValueWitnesses<swift::metadataimpl::NativeB... |
| 34.00 ms | 14.0% | 34.00 ms | __swift_destroy_boxed_opaque_existential_1 Benchmarks |
| 2.00 ms | 0.8% | 0 s | ▼<Unknown Address> |
| 2.00 ms | 0.8% | 2.00 ms | swift::metadataimpl::BufferValueWitnesses<swift::metadataimpl::ValueWitnesses<swift::metadataimpl::NativeB... |
| 54.00 ms | 100.0% | 12.00 ms | ▼-[MessageDispatchTestObjC run] Benchmarks |
| 22.00 ms | 40.7% | 22.00 ms | objc_msgSend libobjc.A.dylib |
| 20.00 ms | 37.0% | 5.00 ms | ▼-[MultiplyByTwo multiply] Benchmarks |
| 15.00 ms | 27.7% | 15.00 ms | -[MultiplyByTwo n] Benchmarks |

Rysunek 4.12: Tablica profilowania dla testów mechanizmów wywołania funkcji: statycznego (u góry), dynamicznego (w środku) i przez wiadomości (na dole)

wydajną formą optymalizacji. Nadal jednak narzut czasowy powodowany przez wywołania funkcji `objc_msgSend` to 22ms, czyli około 40% czasu wykonywania całej metody.

4.7. Złożony algorytm

Wszystkie testy przedstawione w tej pracy skupiały się na przetestowaniu wydajności konkretnych aspektów języka Swift. Ostatni test sprawdza natomiast, jak język Swift sprawdza się w bardziej złożonych zadaniach, takich jak np. algorytm Dijkstry.

Algorytm Dijkstry to metoda znajdowania najkrótszych ścieżek w grafie G o wierzchołkach V i macierzy sąsiedztwa w z pojedynczego źródła s . Pseudokod dla tego algorytmu wygląda następująco:

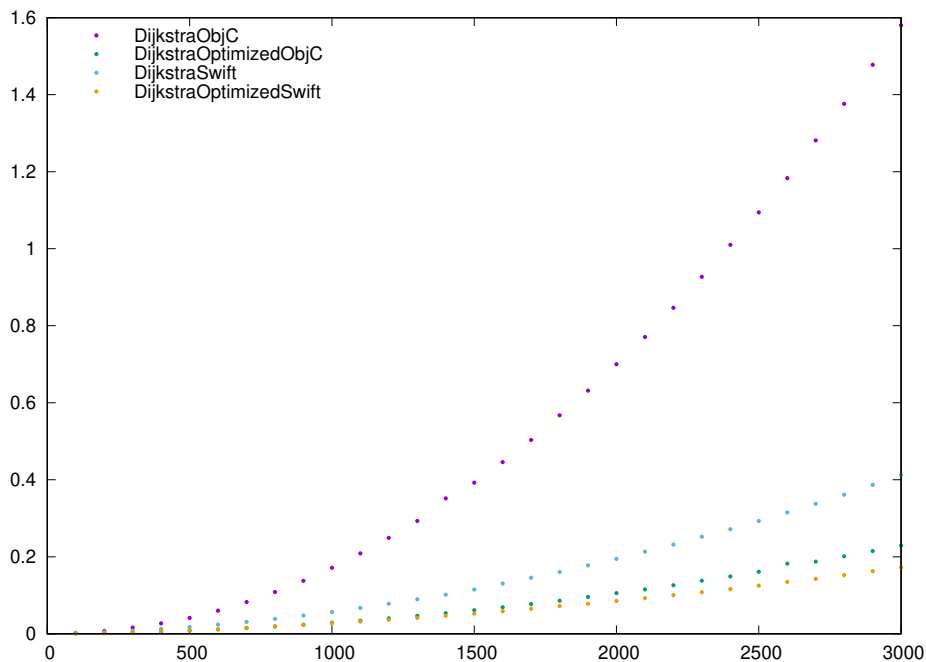
```

proc dijkstra( $V, w, s$ )
     $S = \emptyset$                                 zbiór wierzchołków z obliczoną najkrótszą ścieżką
     $Q = V$                                     kolejka priorytetowa
     $s.d = 0$ 
    for  $v \in Q$  do
         $v.d = +\infty$ 
    end
    while  $Q \neq \emptyset$ 
         $v = Q.removeMin()$ 
         $adj = v.adjacentVertices()$ 
        for  $u \in adj$  do
            if  $v.d + w[v][u] < u.d$ 
                then  $Q.decreaseKey(u, v.d + w[v][u])$ 
            fi
        end
    end
end

```

Ważną częścią algorytmu Dijkstry z punktu widzenia złożoności obliczeniowej jest sposób implementacji kolejki priorytetowej. Ze względu na dużą ilość operacji `removeMin` oraz `decreaseKey`, najwydajniejsze implementacje używają kopców Fibonacciego. Celem tego testu jest jednak porównanie czasu działania jak najbardziej podobnych do siebie implementacji w obu językach, a nie napisanie jak najwydajniejszej implementacji algorytmu, dlatego zdecydowano użyć prostszej struktury danych, czyli zwykłych kopców typu `min`. Pozwoliło to na prostszą analizę czasu działania kodu bez zagłębiania się w szczegóły działania samej struktury.

Algorytm Dijkstry został zaimplementowany dwukrotnie dla każdego z języków. Pierwsze wersje (nazywane dalej *standardowymi*) korzystają ze wszystkich dostęp-



Rysunek 4.13: Wykres czasu działania testów dla algorytmu Dijkstry

nych funkcji danego języka. W przypadku Swift będą to funkcjonalności takie jak protokoły, funkcje wyższego rzędu, takie jak `map` czy `filter` czy funkcje pomocnicze. W wersji dla Objective-C użyto m.in. typów `NSArray` zarządzającego automatycznie długością tablicy, typu `NSNumber` pozwalającego na przechowywanie liczb w tablicy `NSArray` oraz automatycznego zarządzania pamięcią wszędzie tam, gdzie było to możliwe. Kod opisanych powyżej testów znajduje się w plikach (odpowiednio dla Objective-C i Swift) `DijkstraTest.m` oraz `DijkstraTest.swift`. Następnie przeprowadzono analizę działania wyżej opisanych implementacji i na podstawie jej wyników zoptymalizowano implementacje dla obu języków. Porawiony kod znajduje się w plikach `DijkstraOptimized.m` oraz `DijkstraOptimized.swift`.

4.7.1. Analiza działania

Rezultaty poszczególnych testów zostały przedstawione na wykresie 4.13. W przypadku standardowej implementacji przewaga Swifta jest wyraźna, kod napisany w tym języku jest od 3-4 razy szybszy od kodu w Objective-C. W przypadku zoptymalizowanych wersji Swifta kod swiftowy nadal jest szybszy, jednak różnica nie jest już aż tak znacząca i wynosi ok. 30%.

Tablica profilowania została przedstawiona na rysunku 4.14. Test, którego wyniki przestawione są na tablicy został uruchomiony dla grafu o 10000 wierzchołków. Dla lepszej czytelności usunięto wpisy zajmujące nie więcej niż 10ms. Krótka analiza tablicy wskazuje na przyczyny słabej wydajności kodu napisanego w Objective-C. Jak widać, cały test zajął ponad 20 sekund, z czego obliczanie sąsiadnych wierzchoł-

| | | | |
|-----------|--------|-----------|---|
| 20.80 s | 100.0% | 0 s | ▼-[DijkstraTestObjC run] Benchmarks |
| 18.52 s | 89.0% | 382.00 ms | ▼-[DijkstraTestObjC dijkstra] Benchmarks |
| 5.72 s | 27.5% | 391.00 ms | ►-[DijkstraTestObjC adjacentNodesToNode:withMatrix:n:] Benchmarks |
| 4.97 s | 23.9% | 4.97 s | ►-[NSArrayM objectAtIndexedSubscript:] CoreFoundation |
| 2.19 s | 10.5% | 177.00 ms | ►-[NSCFNumber unsignedLongValue] CoreFoundation |
| 1.87 s | 9.0% | 1.87 s | objc_msgSend libobjc.A.dylib |
| 1.56 s | 7.4% | 1.56 s | objc_retain libobjc.A.dylib |
| 1.18 s | 5.6% | 1.18 s | objc_release libobjc.A.dylib |
| 137.00 ms | 0.6% | 137.00 ms | objc_retainAutoreleasedReturnValue libobjc.A.dylib |
| 70.00 ms | 0.3% | 70.00 ms | DYLD-STUB\$\$objc_retainAutoreleasedReturnValue Benchmarks |
| 69.00 ms | 0.3% | 0 s | ►-[PriorityQueueObjC removeMin] Benchmarks |
| 68.00 ms | 0.3% | 0 s | ►-[NSArrayM dealloc] CoreFoundation |
| 58.00 ms | 0.2% | 0 s | ►<Unknown Address> |
| 54.00 ms | 0.2% | 54.00 ms | ►-[NSCFNumber unsignedIntegerValue] CoreFoundation |
| 43.00 ms | 0.2% | 43.00 ms | ►-[GraphElement nodeIndex] Benchmarks |
| 37.00 ms | 0.1% | 37.00 ms | ►-[GraphElement distance] Benchmarks |
| 34.00 ms | 0.1% | 34.00 ms | ►-[GraphElement alreadyComputed] Benchmarks |
| 22.00 ms | 0.1% | 22.00 ms | ►-[DijkstraTestObjC adjacencyMatrix] Benchmarks |
| 21.00 ms | 0.1% | 21.00 ms | ►-[GraphElement queueIndex] Benchmarks |
| 12.00 ms | 0.0% | 0 s | ►-[PriorityQueueObjC moveToTop:] Benchmarks |
| 2.17 s | 10.4% | 2.17 s | objc_msgSend libobjc.A.dylib |
| 65.00 ms | 0.3% | 65.00 ms | DYLD-STUB\$\$objc_retainAutoreleasedReturnValue Benchmarks |
| 45.00 ms | 0.2% | 0 s | ►<Unknown Address> |

| | | | |
|-----------|--------|-----------|---|
| 4.45 s | 100.0% | 0 s | ▼@objc DijkstraTestSwift.run() Benchmarks |
| 2.60 s | 58.4% | 401.00 ms | ▼DijkstraTestSwift.dijkstra() Benchmarks |
| 993.00 ms | 22.2% | 0 s | ►DijkstraTestSwift.adjacentNodes(to:from:) Benchmarks |
| 722.00 ms | 16.2% | 0 s | ►_swift_release_dealloc Benchmarks |
| 182.00 ms | 4.0% | 0 s | ►PriorityQueueSwift.removeMin() Benchmarks |
| 93.00 ms | 2.0% | 93.00 ms | swift_retain Benchmarks |
| 74.00 ms | 1.6% | 0 s | ►specialized ContiguousArray._copyToNewBuffer(oldCount:) Benchmarks |
| 53.00 ms | 1.1% | 53.00 ms | _swift_retain_(swift::HeapObject*) Benchmarks |
| 28.00 ms | 0.6% | 28.00 ms | _swift_release_(swift::HeapObject*) Benchmarks |
| 25.00 ms | 0.5% | 0 s | ►PriorityQueueSwift.updateDecreasedElement(at:) Benchmarks |
| 23.00 ms | 0.5% | 23.00 ms | swift_release Benchmarks |
| 1.19 s | 26.6% | 1.19 s | _swift_retain_(swift::HeapObject*) Benchmarks |
| 550.00 ms | 12.3% | 550.00 ms | _swift_release_(swift::HeapObject*) Benchmarks |
| 47.00 ms | 1.0% | 47.00 ms | swift_isUniquelyReferenced_nonNull_native Benchmarks |
| 32.00 ms | 0.7% | 32.00 ms | swift_retain Benchmarks |
| 21.00 ms | 0.4% | 21.00 ms | swift_release Benchmarks |
| 11.00 ms | 0.2% | 0 s | ►<Unknown Address> |

| | | | |
|-----------|--------|-----------|--|
| 2.50 s | 100.0% | 0 s | ▼-[DijkstraOptimizedTestObjC run] Benchmarks |
| 1.97 s | 78.9% | 307.00 ms | ▼-[DijkstraOptimizedTestObjC dijkstra] Benchmarks |
| 700.00 ms | 28.0% | 700.00 ms | objc_retain libobjc.A.dylib |
| 523.00 ms | 20.9% | 523.00 ms | objc_release libobjc.A.dylib |
| 189.00 ms | 7.5% | 189.00 ms | ►-[NSArrayM objectAtIndexedSubscript:] CoreFoundation |
| 55.00 ms | 2.2% | 55.00 ms | objc_retainAutoreleasedReturnValue libobjc.A.dylib |
| 51.00 ms | 2.0% | 0 s | ►-[PriorityQueueObjC removeMin] Benchmarks |
| 49.00 ms | 1.9% | 49.00 ms | ►-[GraphElement distance] Benchmarks |
| 34.00 ms | 1.3% | 34.00 ms | ►-[GraphElement alreadyComputed] Benchmarks |
| 26.00 ms | 1.0% | 26.00 ms | ►-[GraphElement nodeIndex] Benchmarks |
| 21.00 ms | 0.8% | 21.00 ms | ►-[GraphElement queueIndex] Benchmarks |
| 459.00 ms | 18.3% | 459.00 ms | objc_msgSend libobjc.A.dylib |
| 41.00 ms | 1.6% | 0 s | ►<Unknown Address> |
| 24.00 ms | 0.9% | 24.00 ms | DYLD-STUB\$\$objc_retainAutoreleasedReturnValue Benchmarks |

| | | | |
|-----------|--------|-----------|--|
| 1.63 s | 100.0% | 0 s | ▼@objc DijkstraOptimizedTestSwift.run() Benchmarks |
| 722.00 ms | 44.2% | 424.00 ms | ▼DijkstraOptimizedTestSwift.dijkstra() Benchmarks |
| 156.00 ms | 9.5% | 0 s | ►PriorityQueueSwift.removeMin() Benchmarks |
| 36.00 ms | 2.2% | 0 s | ►PriorityQueueSwift.updateDecreasedElement(at:) Benchmarks |
| 25.00 ms | 1.5% | 25.00 ms | protocol witness for static Equatable, == infix(.,:) in conformance Int Benchmarks |
| 21.00 ms | 1.2% | 21.00 ms | swift_release Benchmarks |
| 17.00 ms | 1.0% | 17.00 ms | _swift_release_(swift::HeapObject*) Benchmarks |
| 16.00 ms | 0.9% | 16.00 ms | swift_retain Benchmarks |
| 461.00 ms | 28.2% | 461.00 ms | _swift_release_(swift::HeapObject*) Benchmarks |
| 426.00 ms | 26.1% | 426.00 ms | _swift_retain_(swift::HeapObject*) Benchmarks |
| 14.00 ms | 0.8% | 14.00 ms | swift_release Benchmarks |

Rysunek 4.14: Tablica profilowania dla testów algorytmu Dijkstry, od góry: standardowy Objective-C, standardowy Swift, zoptymalizowany Objective-C, zoptymalizowany Swift

ków zajęło aż 5,72s (około 27% czasu całego testu). Około 5s zajęło również pobieranie danych z tablic typu `NSArray`. Następnie, wywołanie funkcji `objc_msgSend`, które łącznie zajęło 4,04s (ok. 19% całkowitego czasu). Ostatnim czasochłonnym elementem czasu jest obsługa zliczania referencji. Łączny czas wywołania funkcji `objc_retain`, `objc_release` i ich pochodnych wyniósł łącznie 3,01s, czyli około 14% całkowitego czasu wywołania funkcji. Ciekawy jest znikomo mały czas obsługi kolejki priorytetowej (łącznie około 216ms).

W przypadku implementacji w języku Swift łączny czas testu wyniósł jedynie 4,45s. Największą część zajęła obsługa zliczania referencji - łącznie 2,69s, czyli ponad 60% całkowitego czasu. Obliczanie sąsiednich wierzchołków również okazało się stosunkowo kosztowne i potrzebowało 993ms (22% całkowitego czasu). Obsługa kolejki priorytetowej zajęła 207ms, a więc czas zbliżony do tego z implementacji w Objective-C.

Na podstawie powyższej analizy wyciągnięto wnioski odnośnie elementów implementacji, które należy ulepszyć i przyspieszyć. W przypadku kodu napisanego w Objective-C:

- zmniejszono ilość wywołań funkcji `objc_msgSend` poprzez:
 - włączenie kodu metody `adjacentNodesToNode:withMatrix:n:` do ciała metody `dijkstra`
 - użycie prymitywnego typu `NSUInteger` (typedef na typ `unsigned long` z języka C) zamiast `NSNumber`.
 - użycie prymitywnych tablic z języka C i operacji wskaźnikowych zamiast typu `NSArray` i operatora indeksowania
 - zapamiętanie w zmiennej lokalnej wartości wielkości grafu `n`
- zmniejszono czas obsługi tablicy poprzez użycie typu tablicy znanego z języka C zamiast `NSArray`
- usunięto potrzebę zamiany typu `NSNumber` na typ całkowitoliczbowy poprzez użycie typu `NSUInteger`
- zredukowano czas potrzebny na obsługę liczników referencji poprzez zmniejszenie ilości tymczasowych obiektów (głównie tablic zawierających sąsiednie wierzchołki oraz tymczasowych referencji do liczb opakowanych w typ `NSNumber`)

W przypadku kodu w Swift optymalizacje skupiają się głównie na zmniejszeniu ilości czasu potrzebnego na obsługę licznika referencji:

- włączenie kodu metody `adjacentNodes(to:from:)` do ciała metody `dijkstra`
- użycie pętli `for` zamiast funkcji wyższych rzędów

- użycie wczesnego wychodzenia z pętli za pomocą słowa kluczowego `continue`

Wyniki testów po optymalizacjach są również przedstawione na wykresie 4.13 oraz tablicy profilowania 4.14. Czas wykonania obu testów zmalał, w przypadku implementacji w Swift o 63%, w przypadku Objective-C aż o 0,88%. Wydajność kodu w starszym języku znacznie zbliżyła się do wydajności kodu w Swift. Należy jednak zaznaczyć, że w kilku przypadkach użyto elementów języka C, który jest w pełni kompatybilny z Objective-C i jest jednym z najszybszych języków używanych współcześnie. Można więc pisać bardzo wydajny kod w Objective-C używając funkcjonalności z C, traci się przy tym jednak większość zalet, jakie Objective-C daje (np. ARC, obiektowość).

4.8. Pozostałe testy

4.8.1. Zliczanie słów

Test polega na zliczeniu ilości wystąpień słów w tekście składającym się z n słów. Dla każdej instancji testu generowany jest nowy tekst zawierający słowa wylosowane z tablicy około 100 słów. Kod testów dostępny jest w plikach `WordFrequencyTest.swift` (dla języka Swift) oraz `WordFrequencyTest.m` (dla Objective-C).

4.8.2. Sito Eratostenesa

Test polega na obliczeniu wszystkich liczb pierwszych z przedziału $[2, n)$ przy pomocy metody sita Eratostenesa. Kod testów znajduje się w plikach `SieveOfErathostenesTest.swift` (dla języka Swift) oraz `SieveOfErathostenesTest.m` (dla języka Objective-C).

4.8.3. Zliczanie liter, słów i linii

Test polega na zliczaniu ilości liter, słów oraz linii w tekście długości n znaków. Na potrzeby tego testu przyjęto, że słowo to ciąg znaków zakończony spacją, tabulatorem lub znakiem nowej linii, a linia to ciąg znaków zakończony znakiem nowej linii. Kod testów został zapisany w plikach `CountLinesWordsCharsTest.swift` (dla języka Swift) oraz `CountLinesWordsCharsTest.m` (dla Objective-C).

4.8.4. Konkatenacja napisów

Test polega na n -krotnym skonkatenowaniu napisu *Hello world* ze sobą. Kod testów dostępny jest w plikach `StringConcatenationTest.swift` (dla języka Swift) oraz `StringConcatenationTest.m` (dla Objective-C).

4.8.5. Histogram RGB

Test polega na wygenerowaniu histogramu RGB dla ciągu pikseli o długości n . Każdy piksel zawiera trzy liczby z przedziału $[0, 255]$, opisujące wartość koloru czerwonego, zielonego i niebieskiego. Histogram to dwuwymiarowa tablica wielkości 3×256 . Element na pozycji (i, j) w histogramie opisuje ilość pikseli, które dla koloru i przyjmują wartość j . Kod testów znajduje się w plikach `RGBHistogramTest.swift` (dla języka Swift) oraz `RGBHistogramTest.m` (dla Objective-C).

4.8.6. Szyfr RC4

Test polega na zaszyfrowaniu wiadomości o długości n szyfrem RC4 z kluczem `SecretKey`. Szyfr RC4 to szyfr strumieniowy opracowany w 1987 roku przez Rona Rivesta. Dziś jest uważany za niedostatecznie bezpieczny, jednak przez wiele lat był używany w popularnych protokołach sieciowych, takich jak WEP czy SSL. Kod testów został zapisany w plikach `RC4Test.swift` (dla języka Swift) oraz `RC4Test.m` (dla Objective-C).

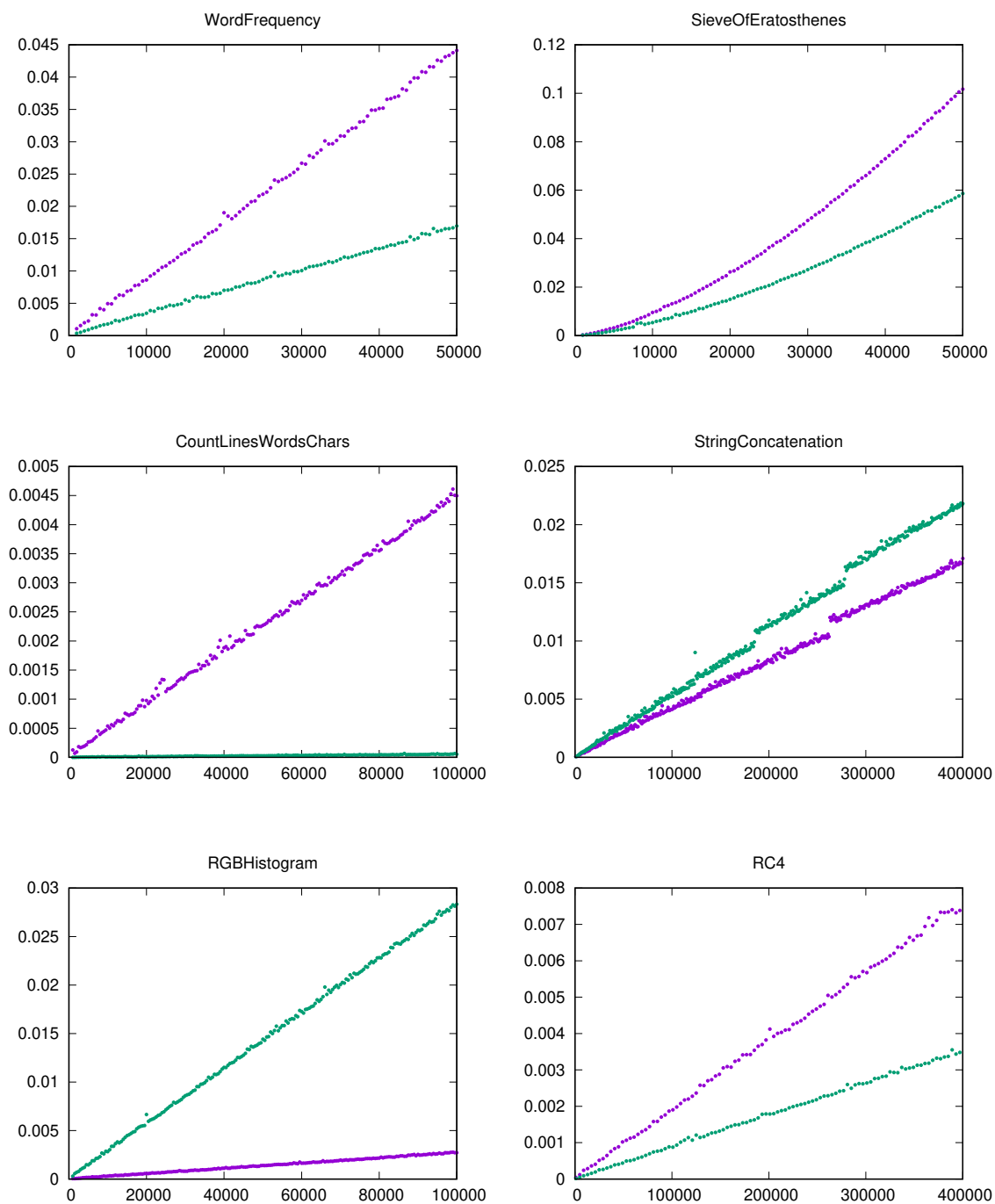
4.8.7. Wyniki pozostałych testów

Wyniki pozostałych testów zostały przedstawione na rysunku 4.15. W czterech testach kod Objective-C był znacząco szybszy. Trzy z nich (`WordFrequency`, `RC4` oraz `CountLinesWordsChars`) wykonywały bardzo dużo operacji na napisach, można zatem wysnuć tezę, że napisy kodowane domyślnie za pomocą standardu UTF-8 powodują duży narzut na wydajność ich przetwarzania, nawet jeśli sam napis zawiera jedynie znaki z tablicy ASCII.

W teście `SieveOfErathostenes` największym obciążeniem były operacje na tablicach. W przypadku implementacji w Objective-C użyto prostych tablic z języka C, stąd tak wysoka wydajność tej implementacji.

Swift okazał się szybszy w dwóch testach. Podobnie jak test `SieveOfErastothernes`, test `RGBHistogram` również polegał na wykonywaniu dużej ilości operacji na tablicy. W tym wypadku do implementacji w Objective-C użyto klasy `NSArray`. Zatem wniosek, jaki nasuwa się na myśl jest następujący: jeśli kod Objective-C ma być szybki, należy użyć struktur danych z C.

Ostatni test `StringConcatenation` pokazał, że implementacja operacji dodawania napisów jest nieco szybsza w nowszym języku.



Rysunek 4.15: Wykresy czasu działania pozostałych testów (zielona linia - Objective-C, fioletowa - Swift)

Rozdział 5.

Wnioski

5.1. Podsumowanie wyników

| L.p. | Test | Względny średni czas testu |
|------|-------------------------------------|-------------------------------|
| 1. | ArrayInsertion | -78,32 % |
| 2. | Fibonacci | -45,84 % |
| 3. | BubbleSort | -87,69 % |
| 4a. | BinarySearchTree - Classic | +129,29 % |
| 4b. | BinarySearchTree - Enums | +103,20 % |
| 4c. | BinarySearchTree - Optimized | -72,09 % |
| 5a. | DispatchMethod - Static vs Message | -75,00 % |
| 5b. | DispatchMethod - Dynamic vs Message | +112,79 % |
| 6a. | Dijkstra | -65,33 % |
| 6b. | DijkstraOptimized | -5,14 % |
| 7. | WordFrequency | +157,48 % |
| 8. | SieveOfEratosthenes | +70,94 % |
| 9. | CountLinesWordsChars | +7718,38 % |
| 10. | StringConcatenation | -22,48 % |
| 11. | RGBHistogram | -90,21 % |
| 12. | RC4 | +116,41 % |

Tablica 5.1: Tabela względnych czasów testów kodu Swift do kodu Objective-C

Tabela 5.1 przedstawia różnicę, między czasem wykonania testu w Swift a Objective-C dla każdego testu opisanego w tej pracy. Wartość ujemna (oznaczona kolorem zielonym) oznacza, że kod Swift był średnio o x procent szybszy w danym teście. Analogicznie wartość dodatnia (oznaczona kolorem czerwonym) wskazuje, że kod Swift był średnio o x procent wolniejszy od kodu Objective-C.

Z dwunastu przeprowadzonych testów, kod pisany w języku Swift był szybszy w

ośmiu przypadkach. W dwóch z ośmiu przypadków wymagane były pewne optymalizacje kodu. W przypadku testu `BinarySearchTree` należało zrezygnować z typów wartościowych na rzecz klas. W teście `DispatchMethod`, dopiero staryczne wywoływanie funkcji okazało się szybsze od wywoływania przez wiadomości z Objective-C. Przykłady te pozwalają wysnuć wniosek, że choć Swift jest ogólnie szybszy od Objective-C, to należy ostrożnie używać bardziej zaawansowanych właściwości tego języka. W czterech testach szybszy był kod Objective-C.

5.2. Wady i zalety Swifta na podstawie testów

Główne cechy Swifta, dzięki którym kod pisany w tym języku jest szybszy to:

- lepiej zoptymalizowane podstawowe struktury danych: `Array`, `Dictionary` oraz `Set` (w tym, możliwość zapisywania prymitywnych typów danych)
- zrezygnowanie z wywoływania metod przez wiadomości i zastąpienie go w większości przypadków wywołaniami statycznymi
- zoptymalizowana obsługa ARC, np. poprzez stosowanie typów wartościowych
- ulepszona obsługa właściwości klas (ang. *properties*)

Aspekty, w których Swift przegrywa wydajnościowo z Objective-C:

- obsługa napisów (ang. *strings*)
- język C jest nadal szybszy od Swifta, zatem dobrze zoptymalizowany kod w Objective-C (tj. w którym najważniejsze fragmenty są napisane w C), może okazać się szybszy od kodu swiftowego

5.3. Inspiracje dla Swift z innych języków programowania

Swift jest językiem stawiającym na bezpieczeństwo, wygodę pracy oraz po części również na szybkość działania. Otwarte źródła języka, dynamiczny rozwój oraz zaangażowana społeczność sprawiają, że ma on szansę podbić nie tylko świat aplikacji na systemy Apple, ale również inne gałęzie branży informatycznej. Aby to jednak miało miejsce, Swift musi poprawić kilka aspektów, w których wyraźnie odstaje od konkurencyjnych języków.

Największym problemem Swifta jest jego młody wiek. Pomimo, że od czasu wydania wersji 1.0 języka minęło już prawie 5 lat, to nadal nie ma stabilnego Interfejsu binarnego aplikacji (ABI), a co za tym idzie, programiści nie mogą używać bibliotek

napisanych w innych wersjach języka niż wersja, w której piszą projekt. Do wersji 3.2 Swift nie miał też kompatybilności wstecznej, przez co przejście na nowszą wersję języka wiązało się z koniecznością konwersji całego kodu, co nie zawsze należało do najłatwiejszych zadań.

Przez pierwsze półtora roku Swift był językiem pozwalającym na tworzenie aplikacji tylko i wyłącznie na platformy iOS i MacOS. Z tego powodu, wszystkie biblioteki w tamtym czasie skupiały się na potrzebach tych dwóch platform. 3 grudnia 2015 roku ogłoszono, że Swift może zostać uruchomiony na systemach rodziny Linux. Okazało się jednak, że poza samym językiem i jego biblioteką standardową nie ma zbyty wielu narzędzi, bo albo nie zostały przystosowane do uruchamiania pod systemem innym niż MacOS, albo w ogóle nie było potrzeby dla takich narzędzi. Społeczność zaczęła wtedy portować istniejące programy i tworzyć nowe, nadal jednak brakuje wielu stabilnych, multiplatformowych bibliotek. Przykładem może być brak multiplatformowej biblioteki do tworzenia interfejsów graficznych, takiej jak Swing czy QT czy zaawansowanego frameworka do tworzenia aplikacji internetowych. Nie istnieje też zintegrowane środowisko programistyczne do pracy w Swift pod systemem Linux.

Testy opisane w tej pracy udowodniły, że automatyczne zliczanie referencji powoduje w niektórych przypadkach duży narzut obliczeniowy. Jeśli język Swift ma się stać językiem do pisania wielkich, wysokowydajnościowych systemów, problem ten musi zostać zaadresowany. Jedną z propozycji na rozwiązanie tego problemu jest implementacja koncepcji własności (ang. *ownership*) pamięci ¹, zastowana ostatnio w języku Rust.

Generyczność w Swift również pozostawia pole do poprawy. W 2016 roku na powstał dokument ², w którym wymienione zostały poprawki, które ulepszyłyby pracę z typami generycznymi, funkcjami generycznymi i protokołami. Znalazły się tam m.in. takie zagadnienia jak:

- *variadic generics*, czyli funkcja znana z języka C++ (tam zastosowana dla szablonów, nie typów generycznych, zasada działania pozostaje jednak taka sama)
- rozszerzenia typów strukturalnych (czyli w wypadku Swift - krotek)
- generyczne protokoły, podobne jak generyczne interfejsy w C# czy Java

Refleksje i wsparcie dla metaprogramingu to kolejne przydatne funkcje, których brakuje w Swift. Co prawda istnieje typ `Mirror` pozwalający na przeglądanie struktury typu (typów zagnieżdżonych, funkcji, właściwości i ich wartości), niemniej w porównaniu z refleksjami z C#, Java, nie wspominając już o językach skryptowych takich jak JavaScript, funkcja ta jest mocno ograniczona. Przykładowo, za

¹<https://github.com/apple/swift/blob/master/docs/OwnershipManifesto.md>

²<https://github.com/apple/swift/blob/master/docs/GenericsManifesto.md>

pomocą mirroringu nie można ustawić wartości właściwości. Brak pełnych refleksji jest sporym utrudnieniem w pisaniu narzędzi automatyzujących pracę, np. generatorów kodu czy linterów.

Twórcy Swifta powinni również poprawić wsparcie dla obliczeń asynchronicznych. W tym momencie, Swift nie zawiera żadnego wbudowanego mechanizmu wykonywania kodu asynchronicznego. Programiści zwykle wykorzystują do tego multiplatformową technologię Grand Central Dispatch, opakowującą wątki systemowe w abstrakcję w postaci kolejek operacji. Już w 2016 roku Chris Lattner wspominał o możliwości zaimplementowania w Swift mechanizmu, znanego z C# czy JavaScript lub modelu aktorów, na razie jednak nie pojawiły się żadne informacje o postępach w tym kierunku.

Ostatnim dużym problemem Swifta jest słaba współpraca z innymi językami programowania. Język Java dzięki platformie JVM może używać kodu napisanego w takich językach jak Kotlin, Clojure, Groovy czy Scala. C# współpracuje np. z C, C++, F#, Visual Basic .NET. Istnieją również implementacje popularnych języków skryptowych takich jak Python czy Ruby pozwalające na kompilowanie ich kodu do kodu pośredniego JVM lub CIL. Swift w wersji 4.0 współpracuje w zasadzie tylko z Objective-C oraz C (jako, że C jest podzbiorem Objective-C). Istnieje również możliwość używania kodu napisanego w C++, wiąże się to jednak z napisaniem biblioteki opakowującej (ang. *wrapper*) w Objective-C.