Spis treści

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 konferecji 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 na język C rozszerzająca go o elementy obiektowe wzrorowane na Smalltalku. Głównym problemem Objective-C jest wysoki poziom trudności. Składnia tego języka różni sie 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 Yalantis [3], 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 Intruments 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 typami powiązanymi (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. pod zmienną można podstawić tylko typ implementujący dany protokół lub dziedziczącym po danej klasie
- automatyczna inferencja typów paremetrów 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 paremetrów 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 przed programistą używającym klasy implementującej dany protokół - typ podstawiany 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 // typ powiązany
    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 powiązanym w Swift

2.3. Typy wartościowe i referencyjne

Podobnie jak w języku C#, typy w Swifcie 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 dla aplikacji używających wielu wątków. Jeśli zmienna typu wartościowego zostanie zadeklarowana jako stała, cały obiekt, łącznie ze wszystkimi polami nie może zostać zmieniony. Typami wartościowymi w Swifcie 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 tego samego adresu w pamięci. Zmienne typu referencyjnego zadeklarowane jako stałe zapewniają jedynie stałość referencji, jednak dane przypisane do zmiennej mogą być bez dowolnie zmieniane. Typami referencyjnymi w Swifcie 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 Swifcie 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 byc 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 domyślnie używa gorliwej ewaluacji wyrażeń, autorzy zaimplementowali jednak dwa rozwiązania pozwalające w podstawowym stopniu na wspieranie leniwych obliczeń. Po pierwsze, w Swifcie, 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 Swifcie jest on zaszyty w samym języku - służy do tego słowo kluczowe *lazy*. Obiekt utworzony przy użyciu tego słowa zostanie stworzony dopiero w chwili pierwszego odwołania się do niego. Drugim rozwiązaniem są leniwe struktury danych, których implementacja opiera się na znanych również z języka C# czy Java generatorach.

```
typealias BigData = Int

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

   // .. reszta ciała klasy
}

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

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

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

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

let evenFibbonacci = Fibbonacci().filter { $0 % 2 == 0 }
var iterator = evenFibbonacci.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 faktu, 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 pozwalanją na definiowanie typów podobnych do algebraicznych typów danych (ang. *Algrebraic data types*) znanych z programowania funkcyjnego.

```
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 za pomocą typu wyliczeniowego

- Dzięki zwięzłej i eleganckiej składni oraz potraktowaniu domknięć na równi klasami i stukturami, 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 niemutowalne struktury danych, co przejawia się w całej składni języka. Dostępne jest słowo odrębne słowo kluczowe `let` służące do deklarowania stałych, parametry przekazywane do funkcji są domyślnie stałymi, a użycie typów wartościowych jest preferowane nad użyciem klas (również w bibliotece standardowej).

- Swift posiada zaawansowany mechanizm *pattern matchingu*, który można wykorzystać do przetwarzania 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`.

```swift
// 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, dodano dużą ilość dodatkowej składni upraszczającej kod (tzw. *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 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 Swifcie zotały 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

```swift
// 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 Swifcie), wspierają dziedziczenie i protokoły. 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(@"> Tytul: %@", book.title);
    NSLog(@"> Autor: %@", book.author);
    NSLog(@"> Ilosc 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("> Tytul: \(book.title)");
        print("> Autor: \(book.author)");
        print("> Ilosc 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 Swifcie

3.1.2. Zarządzanie pamięcią

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, odciążajaca 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 wcześniej. 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 Swifcie. Główną różnicą jest jednak sposób implementacji. W Objective-C, ARC jest rozszerzeniem języka, opierającym się głównie na generowaniu kodu odpowiedzialnego za zliczanie referencji. W Swifcie natomiast, ARC jest właściwością języka, 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, niskopoziomowych bibliotek 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 Swifcie 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 ciągów znakowych, dodano dużo bardziej czytelną 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 zupełnie 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 Swifcie natomiast, ARC jest podstawowym elementem języka posiadającym odrębną składnię i słowa kluczowe.

3.2.3. System typów

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

- w Swifcie w zasadzie nie występuje niejawne rzutowanie typów. Każde (nawet najprostsze, jak rzutowanie z typu całkowitoliczbowego na zmiennoprzecinknowy) 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 do obiektu. Z tego względu rozwiązywanie, którą funkcję należy wywołać dzieje się w metodzie `objc_msgSend` w trakcie działania programu. W Swifcie natomiast wywołania metod opierają się na wskaźnikach do metod oraz tablicy *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 `nil`
- 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 Swifcie 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 większa
- automatyczne sprawdzanie przekroczenia zakresu liczb całkowitych
- zmiany w składni, takie jak: obowiązkowe nawiasy {...} dla ciał funkcji warunkowych i pętl, obsługa wszystkich możliwych wartości dla typów wyliczeniowych, konieczność zwrócenia wartości w fukcjach, które deklarują zwracany typ, brak instrukcji `goto` itp.

Rozdział 4.

Testy

4.1. Założenia i platforma testowa

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 conajmniej 10 pomiarów, wyniki przedstawione na wykresach są średnimi tych pomiarów.

Każdy test został zaimplementowany w conajmniej 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
- (opcjonalnie) dodatkowych testach sprawdzających wnioski wyciągnięta podczas analizy wyników

Kod testów, razem z platformą potrzebną 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 przeprowadzone na laptopie Macbook Pro o następującej specyfikacji:

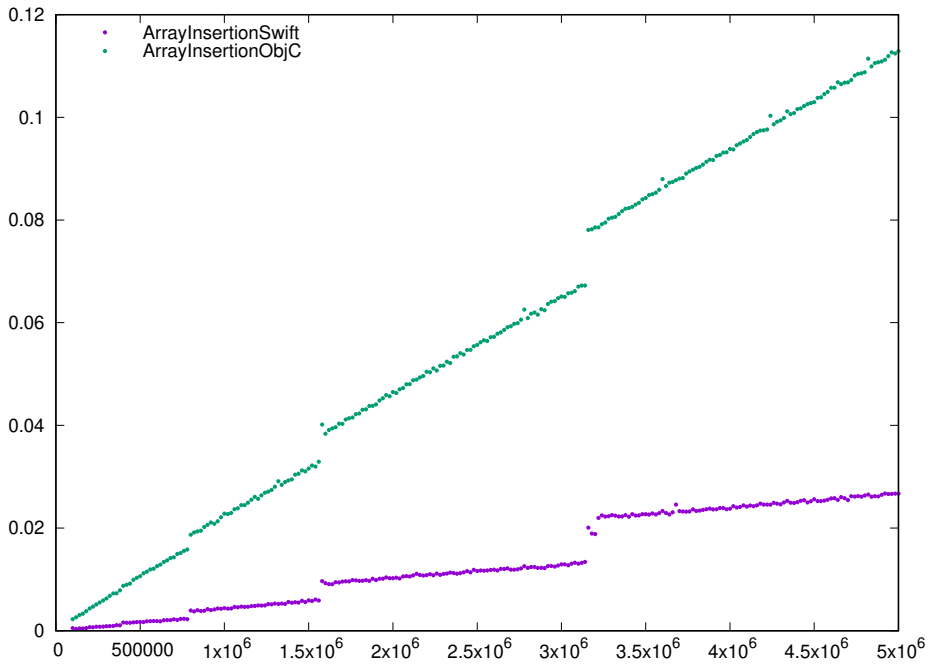
- procesor: Intel Core I7-7700HQ 2,8 GHz
- pamięć RAM: 16 GB LPDDR3
- system operacyjny macOS High Sierra 10.13.1

- 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)

4.2. Wstawianie elementu do tablicy

Test polega na dodaniu do pustej tablicy `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 wynika, że w obu językach dodawanie elementów do tablicy jest operacją działającą w czasie asymptotycznie liniowym, w języku Swift jest on jednak czterokrotnie krótszy.

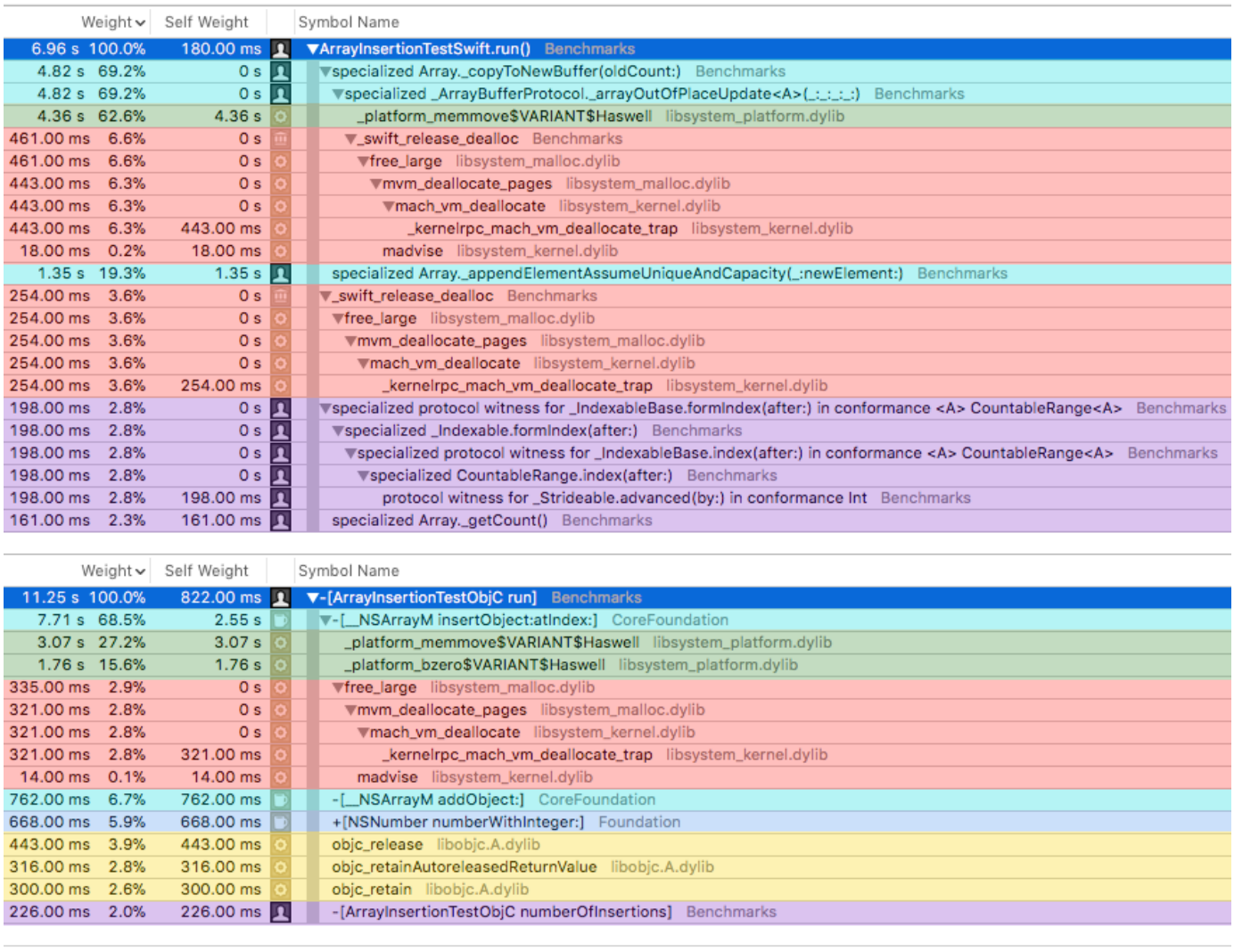
Tak jak większość popularnych języków, zarówno Objective-C, jak i Swift zapewniają ciągłość pamięci tablicy. Z wykresu wynika, że oba języki rezerwują za wczasu większą ilość pamięci niż jest rzeczywiście potrzebna, a w razie przekroczenia już zajętej pamięci zwiększają rozmiar eksponencjalnie (na wykresie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



Rysunek 4.1: Wykres czasu działania testu ArrayInsertionTest dla obu języków

4.2.1. Analiza działania

Rysunek 4.2 przestawia 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 Swift. Tablica profilowania wskazuje potencjalne powody takiego stanu:



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 numberWithInteger:] 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)

- 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 pętli `for` przebiegającej od 0 do `numberOfInsertions`. W związku z tym można założyć, że czas działania pętli `for` to czas działania funkcji `run` (oznaczony na tablicy profilownia 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 klasycznej 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 pętle w języku Swift wydają się być bardziej skomplikowane, są wydajniejsze niż proste pętle `for` z Objective-C.

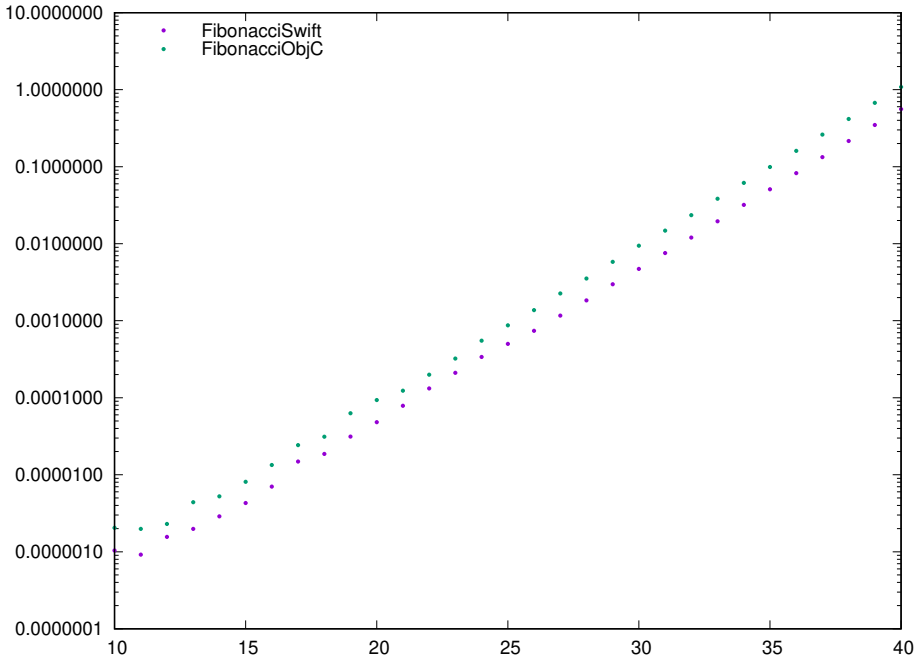
- W Objective-C kod funkcji `addObject` został zinline'owany do wywołań metod `insertObject:atIndex` oraz `addObject` klas wewnętrznych. Łączny czas wywołań tych dwóch metod to 3,31s. W Swift metoda `append` również została zinline'owana do metod prywatnych `_copyToNewBuffer(oldCount:)` oraz `_appendElementAssumeUniqueAndCapacity(_:newElement:)`, 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) zajmuje podobny czas w obu językach (4,36s dla Swift i 4,82 dla Objective-C). Wynika to z faktu, 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).

- 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, takie jak `objc_retain` czy `objc_release` (kolor zółty na rysunku). Jak widać, ich wywołania zajmują około 10% czasu wykonywania całego programu (1,05s)

- 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).



Rysunek 4.3: Wykres czasu działania testu FibonacciTest dla obu języków

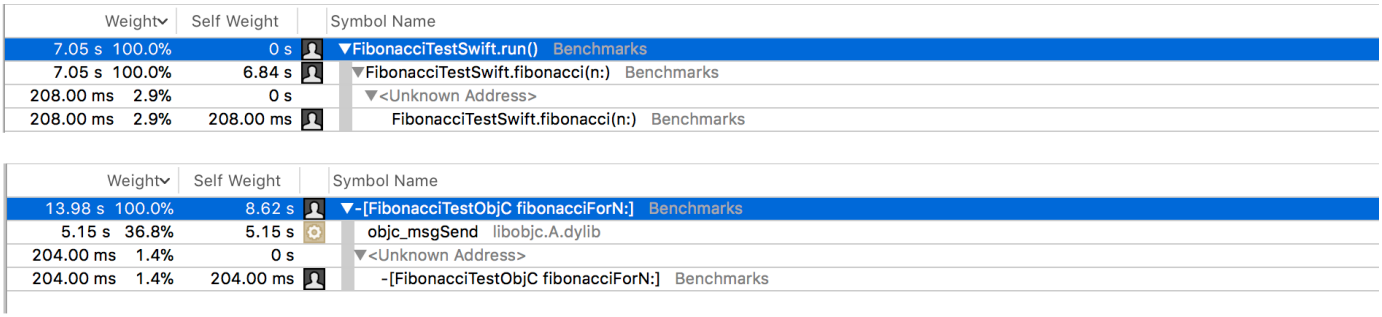
4.3. Rekurencyjne obliczanie liczby Fibonacciego

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 Objective-C.

4.3.1. Analiza działania

Tablica profilowania 4.4 obu programów pozwala znaleźć powody różnicy prędkości pomiędzy implementacjami. 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 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.



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

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

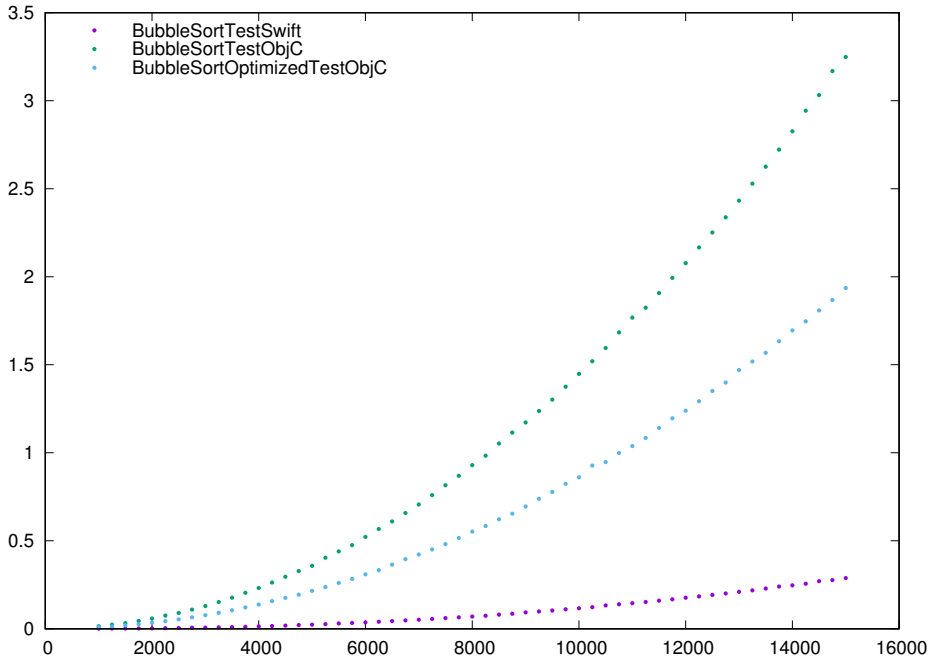
Jak wynika z wykresu, implementacja w Swift jest ponad 10-krotnie szybsza niż najprostsza implementacja w Objective-C. Posortowanie tablicy 15000 elementów zajmuje tylko $288ms$ w przypadku kodu w Swift i $3,2s$ dla Objective-C.

Rysunek 4.6 przedstawia tablicę profilowania testu `BubbleSort` zaimplementowanego w języku Swift (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)
- wywołanie operacji zdefiniowanych w samej metodzie `BubbleSort(:_)` zajęło około $800ms$. Te operacje to obługa 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 $4,58s$.



Rysunek 4.5: Wykres czasu działania testu BubbleSortTest dla obu języków

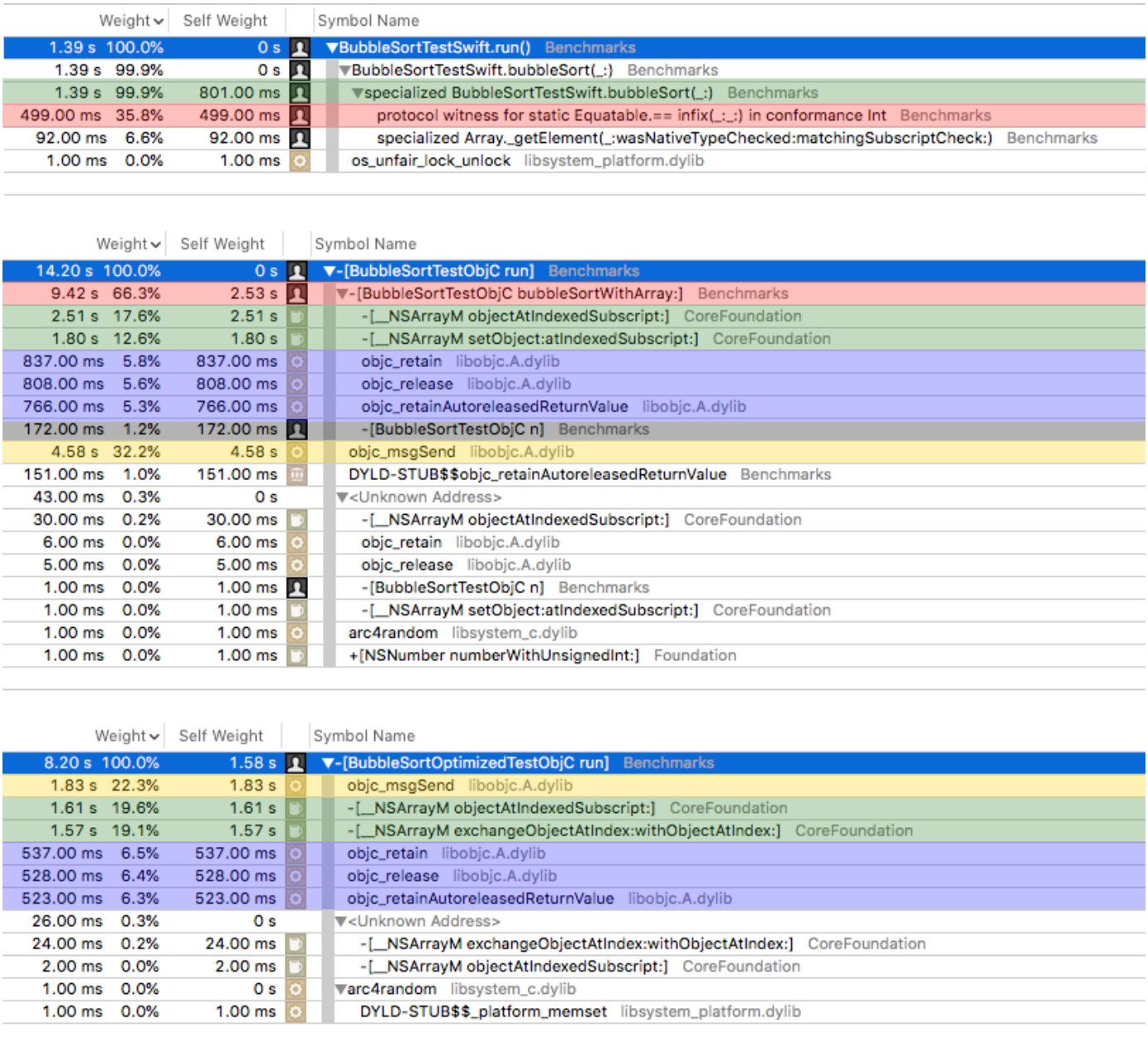
Drugą najbardziej czasochłonną grupą opeacji były operacje odczytu i zapisu do/z tablicy (kolorem zielony na tablicy profilowania). Metody `objectAtIndexedSubscript:` i `setObject:atIndexedSubscript` są wywoływane po użyciu operatora indeksowania na tablicy `NSArray` i łącznie czas ich wywołania zajmuje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

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 własciwości `n` . W Objective-C dla każdej właściwości o nazwie `x` kompilator generuje:

- zmienną o nazwie `_x`
- getter o nazwie `x`
- setter `setX:`

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



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

Z powyższej analizy można łatwo wyznaczyć fragmenty kodu, które powinny zostać zoptymalizowane w implementacji w Objective-C:

- 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 jej wartość 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.

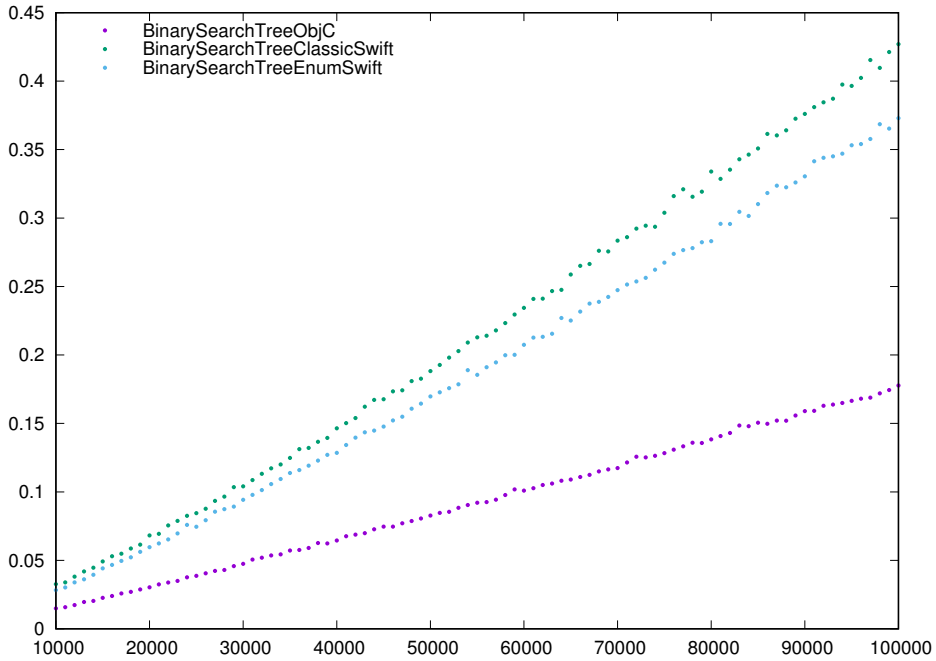
Największy przyrost wydajności zaobserwowano dla wywołań funkcji `objc_msgSend`, czas $4,58s$ udało się zmniejszyć do $1,83s$ (spadek o 60%). Główne powody to zastąpienie kilku wywołań operatora indeksowania jednym wywołaniem metody zamieniającej elementy miejscami oraz trzymanie referencji na tą metodę 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 implemenacją tego problemu. Zdefiniowana została struktura `Node` przechowująca wskaźniki na lewe i prawe poddrzewo oraz wartość przechowywaną w węźle.

Nieco ciekawiej wygląda kwestia implementacji algorytmu budowania drzewa w języku Swift. Po pierwsze, programości Swift często przedkładają bezpieczeństwo



Rysunek 4.7: Wykres czasu działania testu BinarySearchTree

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 typy pierwszoklasowe 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

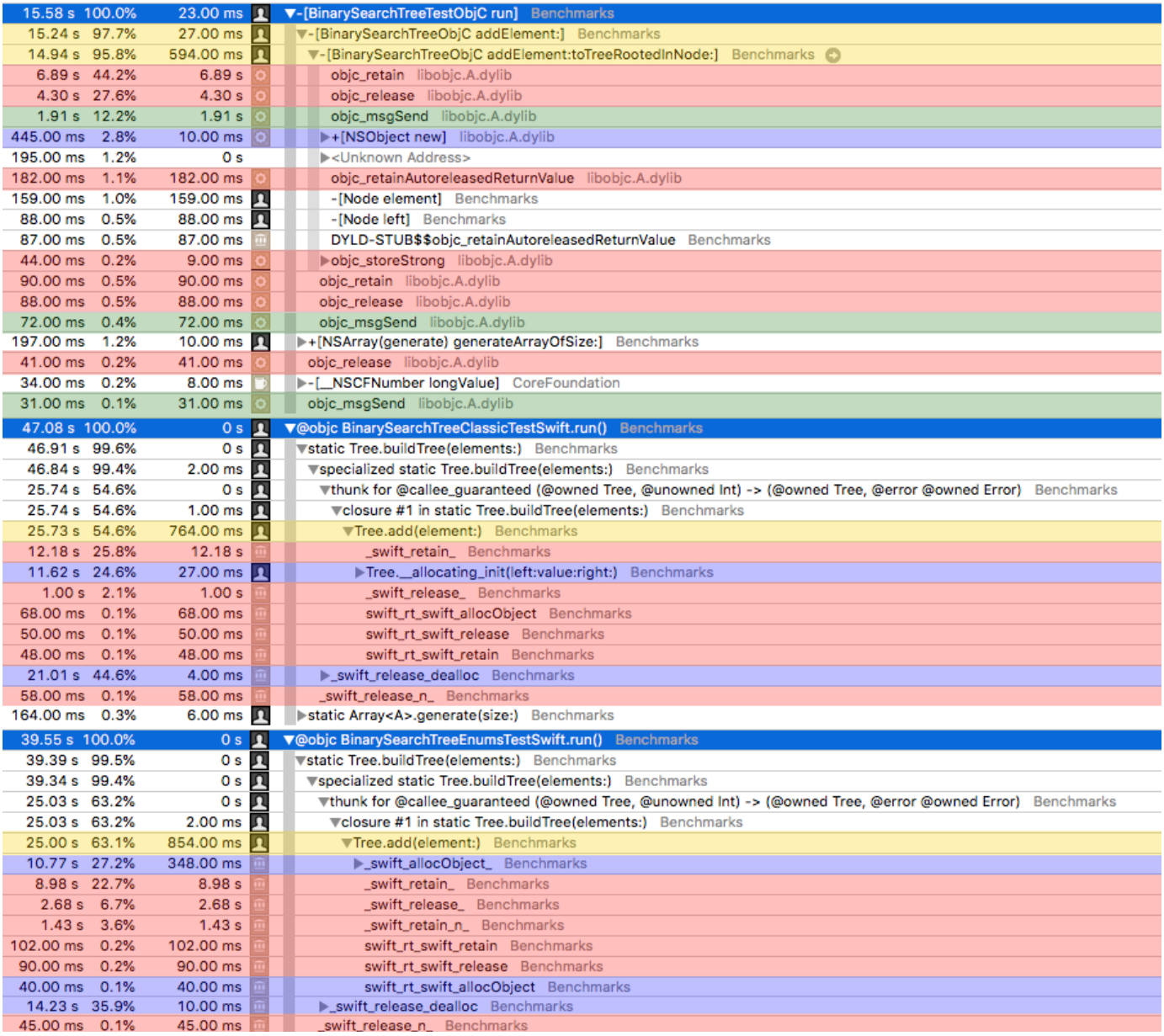
Wykres 4.7 Przedstawia wyniki działania testu tworzącego drzewo binarne o wielkości `n` . Jest to pierwszy test, dla którego implementacja w Objective-C jest szybsza od implementacji w Swift. Kod napisany w starszym z języków wykonuje się około 2 razy szybciej od implementacji napisanej Swift przy pomocy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

Rysunek 4.8 przestawia 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 dla implementacji w Swift.

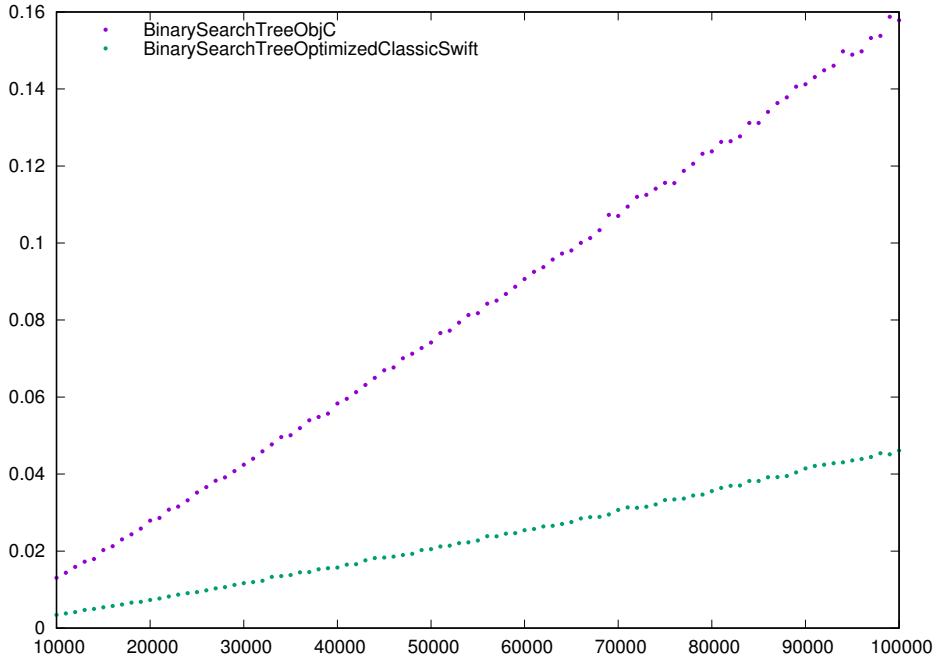
Największą ilość czasu podczas wykonywania testów w Swift zajęło tworzenie nowych obiektów (kolor niebieski na tablicy profilowania) oraz obsługa ARC (kolor czerwony). Powodem, dla którego tak się działo jest wielokrotne tworzenie obiektów dla tych samych węzłów, co powodowało bardzo częste wywoływania inicjalizatora i deinicjlizatora oraz w późniejszym etapie funkcji odpowiadających za inkrementację i dekrementację licznika referencji. Dla implementacji w Swift z użyciem typów wyliczeniowych łączny czas wywołań inicjalizatorów i deinicjalizatorów wyniósł 25,04s, a operacje obługi 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 kodu 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 wyraźnie 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%).



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	▼thunk 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

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 w języku, który nie jest w pełni do tego przystosowany.

4.6. Rodzaje wywołania metod

Język Swift obsługuje dwa rodzaje wywołań metod: statyczną (ang. *direct dispatch* lub *static dispatch*) i dynamiczną (ang. *table dispatch* lub *dynamic 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 jedynym sposobem wywoływania w starszym z języków.

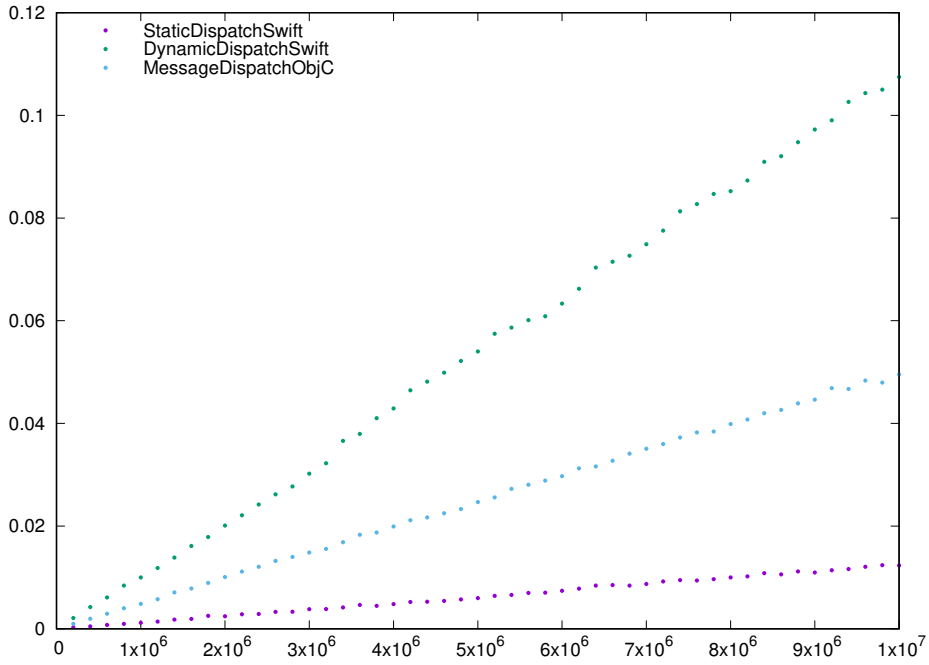
Wywoływanie statyczne jest uznawane za najszybszy sposób wykonywania metod. Główną cechą jest pełna wiedza o tym, która metoda ma zostać wykonana już w trakcie procesu kompilowania. Z tego powodu, kompilator może stosować dużą ilość optymalizacji, takich jak bezpośrednie odwołania do adresu funkcji czy wywołanie funkcji "w linii"(ang. *inline*). Główną jej wadą jest natomiast 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 w trakcie działania programu. Uniemożliwia też stosowanie niektórych metod optymalizacji kodu. W Swift wywoływanie 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



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

języka, takie jak selektory 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 sie 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 tablcy profilowania) oraz wywołanie funkcji `multiply` . To właśnie brak dodatkowych kosztownych operacji powoduje, że wywołanie statyczne jest najszybszym z trzech porównywanych mechanizmów.

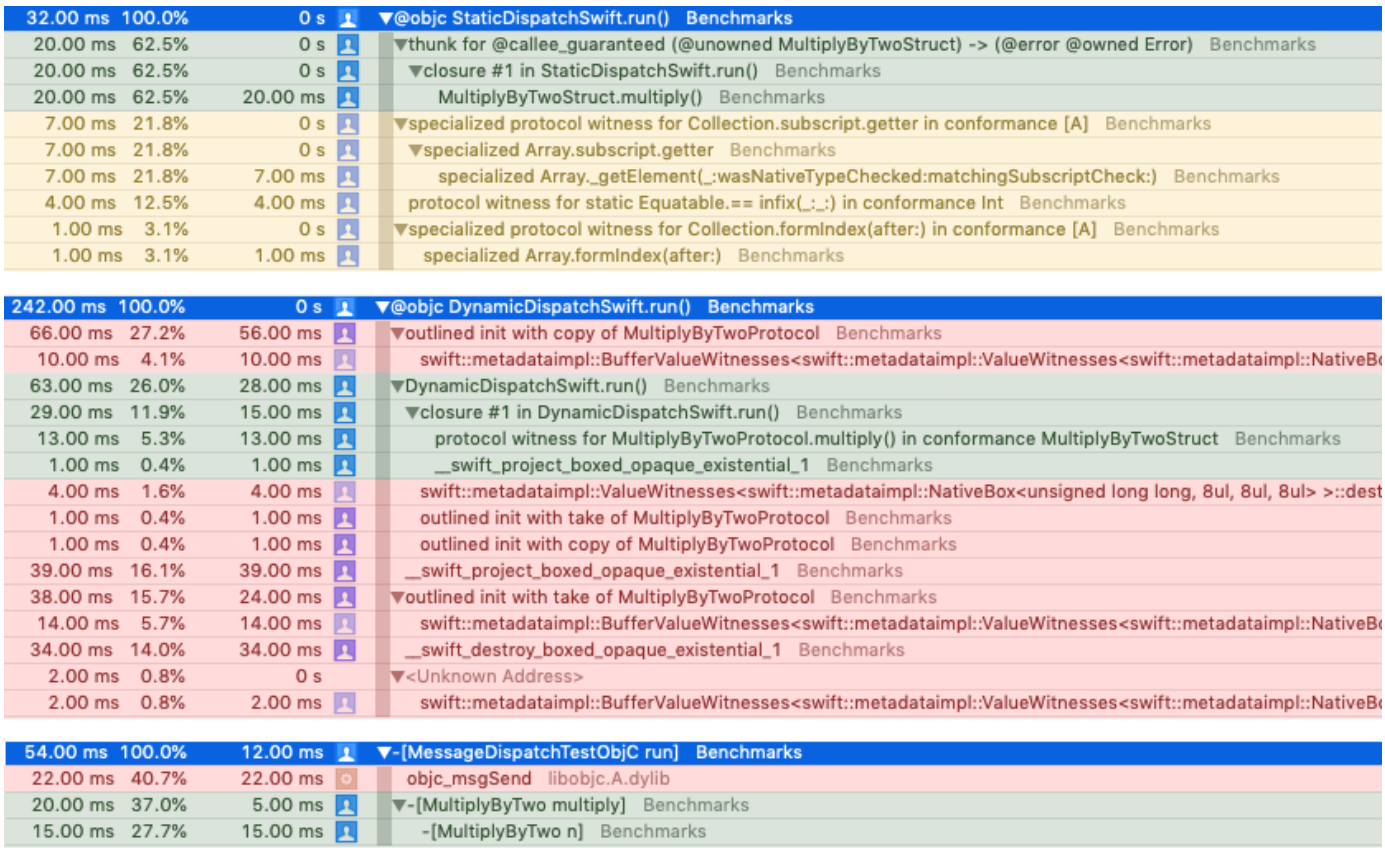
Przy wywołaniu dynamicznym, implementacja funkcji `multiply` jest schowana za protokołem. Z tego powodu, kompilator musi w trakcie działania programu znaleźć adres do odpowiedniej implementacji, którą należy wywołać. Operacja ta zostanie wykonana w dwóch krokach:

- 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 przykładzie znajdzie się tam adres implentacji dla funkcji `multiply` w strukturze `MultiplybyTwoStruct` .

- następnie kompilator powiąże utworzoną tablicę z danym obiektem struktury tworząć 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 wydajną formą optymalizacji. Nadal jednak narzut czasowy powodowany przez wywołania funkcji `objc_msgSend` to 22ms, czyli około 40% czasu wykonywania całej metody.



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

4.7. Złożony algorytm

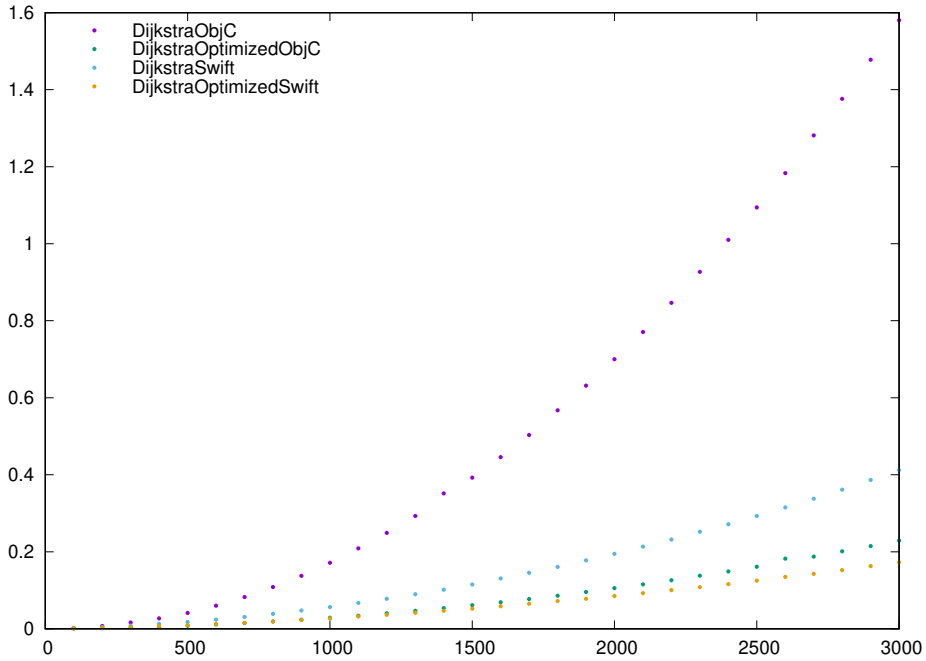
Wszystkie testy przedstawione w tej pracy skupiały sie na przetestowaniu wydajności konkretnych aspektów języka Swift. Ostatni test sprawdza natomiast, jak język Swift sprawdza się w bardziej złożonych zadaniach, takich jak np. algorytm Dijkstry.

Algorytm Dijkstry to metoda znajdowania najkrótszych ścieżek w grafie. Nie dany będzie graf G o wierzchołkach V, macierzy sąsiedztwa w oraz pojedynczym źródle s. Pseudokod dla tego algorytmu wygląda następująco:

```
proc dijkstra(V, w, s)
  S = ∅                                  zbiór wierzchołków z obliczoną najkrószą ścieżką
  Q = V                                                          kolejka priorytetowa
  s.d = 0
  for v ∈ Q do
      v.d = +∞
  end
  while Q ≠ ∅
        v = Q.removeMin()
        adj = v.adjacentVertices()
        for u ∈ adj do
            if v.d + w[v][u] < u.d
              then Q.decreaseKey(u, v.d + w[v][u])
            fi
        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 się 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 dostępnych 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 przechowywa-



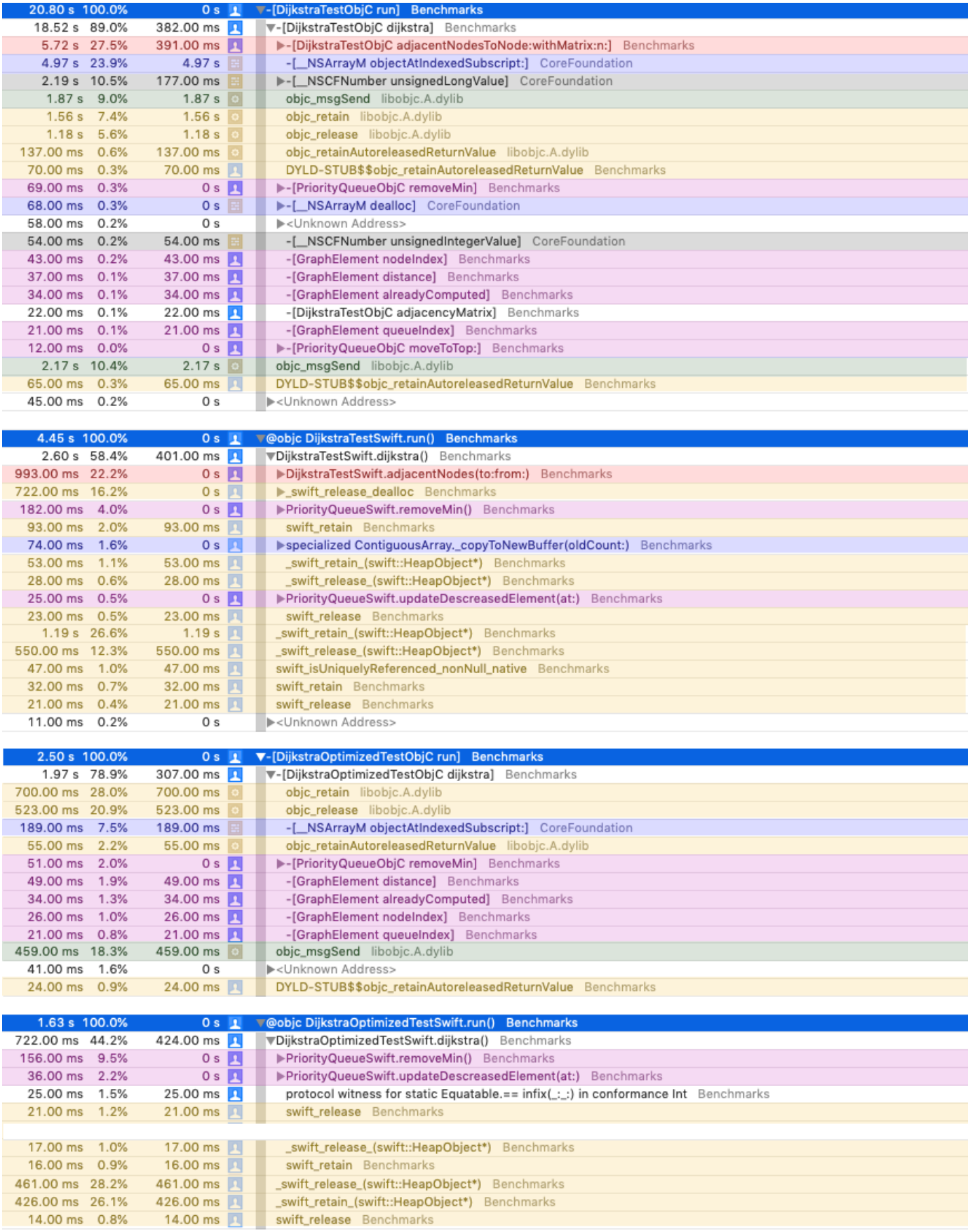
Rysunek 4.13: Wykres czasu działania testów dla algorytmu Dijkstry

nie liczb w tablicy `NSArray` oraz automatycznego zarządzania pamięcią wszędzie tam, gdzie było to możliwe. Kod opisanych powyżej testów znajduje się w plikach `DijkstraTest.m` (dla Objective-C) oraz `DijkstraTest.swift` (dla Swift). Nastepnie 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 3-4 razy szybszy od kodu w Objective-C. W przypadku zoptymializowanych 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 sasiednich wierzchołków zajęło aż $5,72s$ (około 27% czasu całego testu). Również około $5s$ również pobieranie danych z tablic typu `NSArray` . Następnie, wywołanie funkcji `objc_msgSend` , które łącznie zajęło $4,04s$ (ok. 19% całkowitego czasu). Ostatnim czasochłonnym elementem jest obługa zliczania referencji. Łączny czas wywołania funkcji `objc_retain`



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

, `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 Swfit 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 Obiective-C:

- zmniejszono ilość wywołań funkcji `objc_msgSend` poprzez:
  - włączenie kodu metody `adjacentNodesToNode:withMatrix:n:` do ciała metody `dijkstra`
  - użycie primitywnego typu `NSUInteger` (alias na typ `unsigned long` z języka C) zamiast `NSNumber` .
  - użycie prymitywnych tablic z języka C i operacji wskźnikowych 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 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 wylosowany 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 wielokści 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 `RC4Text.m` (dla Objective-C)

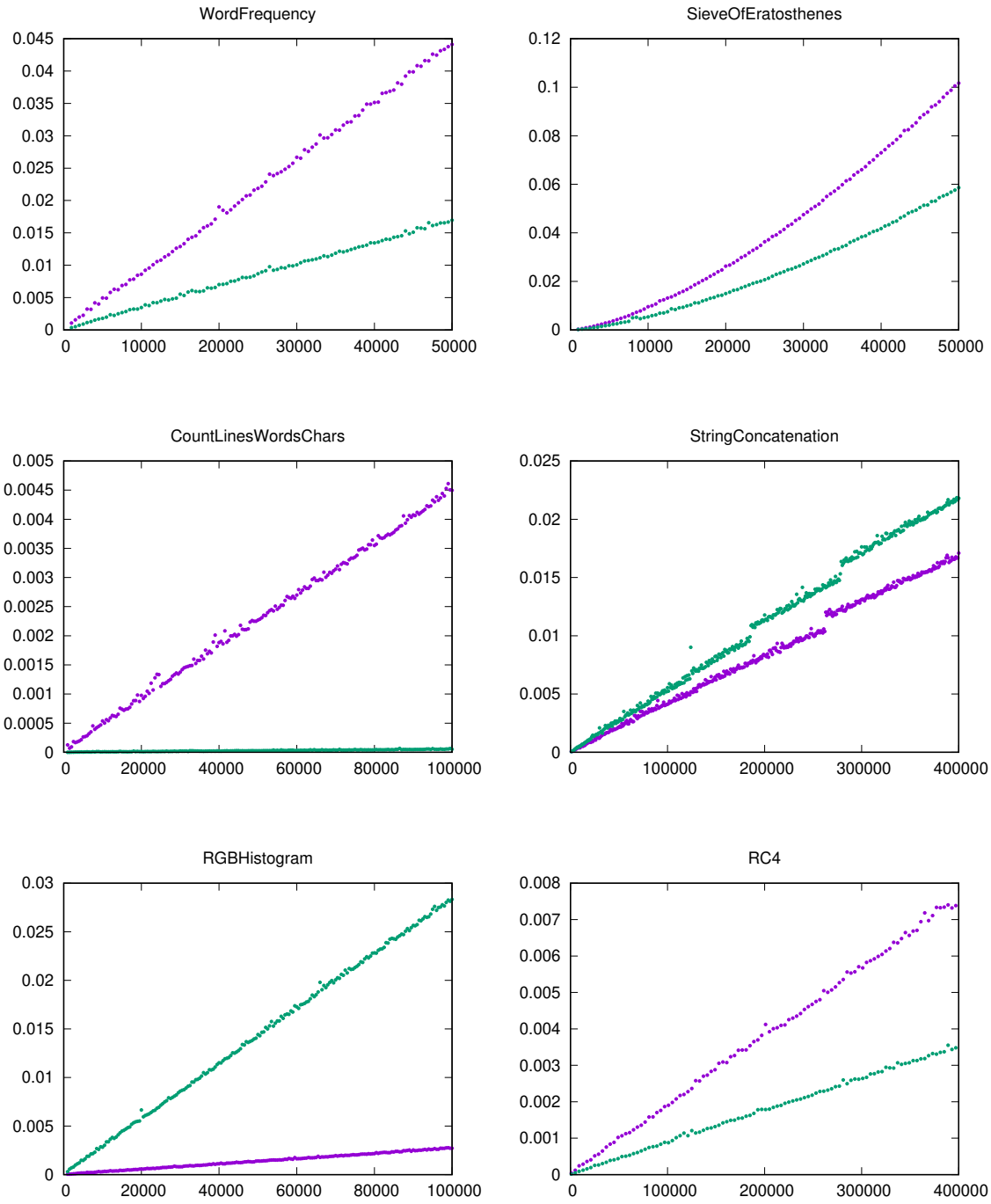
4.8.7. Wyniki pozostałych testów

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 `SieveOfErastothenes`, test `RGBHistogram` rówież 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 zielony) 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 statyczne 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 czterach 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 przechowywania w nich 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:

- obługa napisów (ang. *strings*)
- język C jest nadal szybszy od Swift, 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 bezpiczeństwo, wygodę pracy oraz po cześci również na szybkość działania. Otwarte źródła języka, dynamiczny rozwój oraz zaangażowana społeczość 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 (łącznie z bibliotekami zewnętrznymi), 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 zbyt wielu narzędzi, gdyż albo nie zostały przystosowane do uruchamiania pod systemem innym niż MacOS, albo do tej pory nie było potrzeby, żeby stworzyć takie narzędzia. 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 rozbudowanych, 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 do generycznych interfejsów w C# czy Java

Refleksje i wparcie 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 posiada 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 `async/await`, 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 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 kod C (ang. *wrapper*) w Objective-C.