УНИВЕРЗИТЕТ УНИОН

РАЧУНАРСКИ ФАКУЛТЕТ

Функционално програмирање у Swift језику

Ментор: Проф. Др Душан Тошић

Аутор: Ивица Миловановић

1 Увод ... 3
1.1 Функционално програмирање ... 3
1.2 Swift језик ... 5
2 Преглед функционалних одлика Swift језика ... 9
2.1 Функције као типови првог реда ... 9
2.2 Структуре, као типови вредности ... 10
2.3 Тестирање подударања образаца (Pattern Matching) ... 11
2.4 Уније дискриминатора ... 13
2.5 Генерички типови и протоколи ... 14
2.6 Оператори ... 14
2.7 Методи, глобалне функције и оператори ... 15
3 Функције вишег реда ... 16
3.1 Функције вишег реда уграђене у стандардну библиотеку ... 16
3.2 Дефиниција и примена додатних функција вишег реда, присутних у F# језику ... 16
3.3 Парцијална апликација ... 22
3.4 Композиција функција ... 23
3.5 Алтернативни приступи композицији у Swift језику ... 24
4 Функтори, монаде и апликативни функтори ... 32
4.1 Дефиниција ... 32
$F(f) : F(X) \to F(Y$ ... 32
4.2 Имплементација у програмском језику Haskell ... 32
4.3 Имплементација у програмском језику F# ... 34
4.4 Монаде у језику Swift ... 34
5 Дизајн фунционалне библиотеке за компрехенсију листа ... 41
6 Закључак ... 54
Библиографија ... 54

1 Увод

1.1 Функционално програмирање

Иако функционално програмирање тек у последњих неколико година улази у главне токове софтверске индустрије, у питању је веома стара парадигма. Први функционално орјентисани језик, Lisp, направљен је педесетих година 20. века, готово у исто време кад и Fortran. Lisp и остали функционални језици инспирисани су ламбда рачуном, којег је развио Алонсо Черч, покушавајући да на њему заснује читаву логику [1]. Тај преамбициозни циљ није остварен, али се ламбда рачун показао као успешан модел израчунљивости у рачунарским наукама. Тјуринг је показао да је ламбда рачун подједнако изражајан као Тјурингова машина, на којој се заснива процедурално програмирање [2]. Централни концепт ламбда рачуна је израз, који може бити:

1) Променљива x;
2) Ламбда израз, тј. анонимна функција, $\lambda x.x$;
3) Апликација функције $x\ y$

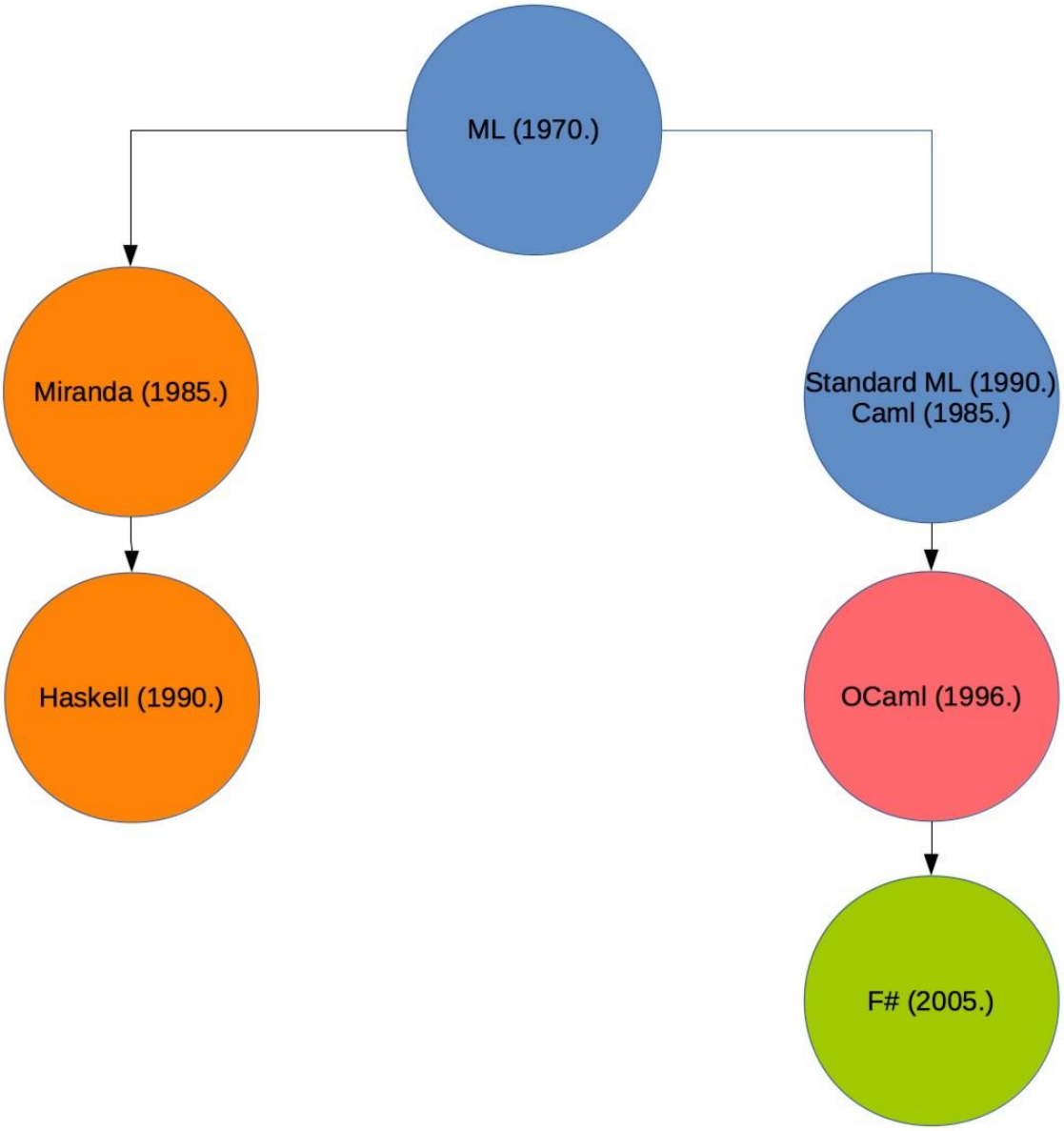
Показано је да се сваки програм може написати као низ таквих израза. Сваки ламбда израз има тачно један аргумент. У пракси је, наравно, неопходно користити функције више аргумената. Ово се постиже тако што се, као резултат ламбда израза, враћа нови ламбда израз, са другим аргументом:

$$(\lambda x.(\lambda y.x\ y))$$

У практичном функционалном програмирању се ово постиже коришћењем Каријевих функција (currying) и парцијалном апликацијом, што описујемо у делу 3.3. Ламбда изрази могу, поред враћања функција као резултата, да узимају функције као аргументе. Такви ламбда изрази називају се функцијама вишег реда, изузетно су важни за функционално програмирање и о њима детаљно говоримо у делу 3.

Као што смо рекли, Lisp језик представља прву примену ламбда функција у рачунарским наукама. Поред тога, овај језик је изузетно значајан јер је у рачунарске науке увео концепте без којих се савремени рад не може замислити, као што су структуре података, аутоматско управљање меморијом, рекурзија итд. За тему овог рада је значајна развојна линија која започиње језиком ML, седамдесетих година прошлог века, јер језици Haskell и F# припадају овој линији. За разлику од директних наследника ML-а, од којих је најмлађи F#, Haskell припада грани чистих и лењих функционалних језика [3], као што је приказано на Слици
1.

Многи концепти, којима се бавимо у овом раду, као што су генеричко програмирање (параметарски полиморфизам), закључивање (инференција) типа, алгебарски типови података (уније дискриминатора), тестирање подударања образаца (Pattern Matching), уведени су кроз језик ML. Најкоришћенији савремени дијалекти овог језика су Standard ML и Caml. Деведесетих година прошлог века развијен је језик OCaml, којим се проширује Caml, додавањем конструктата за објектно орјентисано програмирање. Језик F#, на кога се често позивамо у



Слика 1. Развојно стабло језика инспирисаних ML-ом

овом раду, је дијалект OCaml језика, с том разликом што F# користи објектни модел .NET платформе. Друга грана ML стабла започиње језиком Miranda. Miranda преузима многе одлике ML-а, уз две битне разлике - све функције у Miranda језику су чисте и евалуација је лења. Након објављивања, Miranda је инспирисала плодна истраживања на пољу сличних функционалних језика. Ради координације тих истраживања, група истраживача, предвођена Сајмоном Пејтоном Џонсом, објављује језик Haskell, који постаје de facto стандардни функционални језик. Поред опште намене, језик служи као истраживачки модел за тестирање нових концепата у рачунарским наукама. Бројни концепти који су зачети у Haskell-у, постали су саставни део комерцијалнијих језика, као што је C#. Добар показатељ истраживачке и практичне важности Haskell-а и функционалног програмирања уопште, је то да компанија попут Microsoft-а финансира таква истраживања. Сајмон Пејтон Џонс је од 1998 запослен у истраживачкој лабораторији те компаније, а резултати његовог тима су допринели развоју .NET

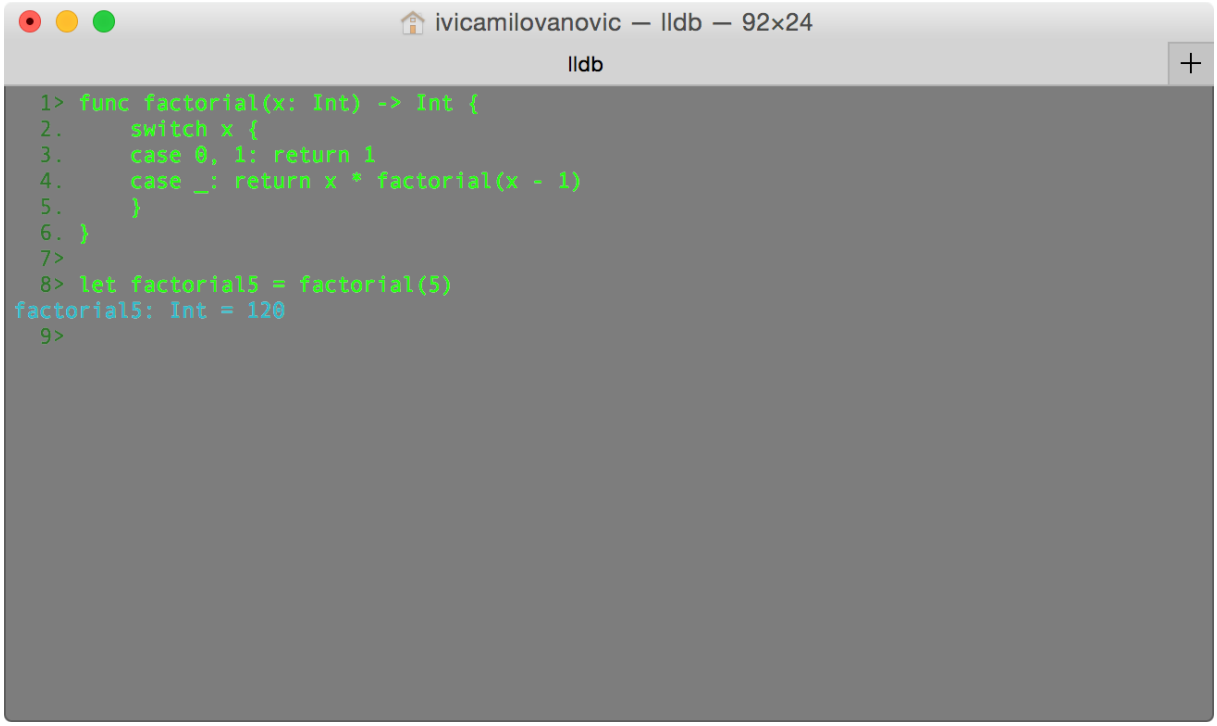
платформе и C# и F# језика. C# није једини индустријски језик који поседује одлике преузете из функционалних језика. Чак и они језици, као што су Java и C++, који су традиционално коришћени готово искључиво за процедурално и објектно орјентисано програмирање, у својим новијим верзијама добијају ламбда изразе, функције првог реда и сличне одлике, те тиме постају знатно погоднији за имплементацију функционалног кода. Swift језик, који је централна тема овог рада, део је управо тог тренда. Али, за разлику од Java-е и C++, у Swift-у је функционално програмирање подржано од прве верзије, те је сасвим природно уклопљено у остатак језика.

1.2 Swift језик

Swift језик [4] је развијен у компанији Apple, са наменом да постане главни језик за развој софтвера на iOS и OS X платформама. Да би се разумела потреба за новим језиком, потребно је најпре рећи нешто о Objective-C, тренутном језику за развој апликација на ове две платформе. Objective-C је развијен осамдесетих година прошлог века, као строги надскуп C језика, коме су додата својства неопходна за објектно орјентисано програмирање. Објектно орјентисани слој је инспирисан језиком Smalltalk. Objective-C је својевремено представљао револуционарни напредак, омогућивши далеко једноставније програмирање графичког интерфејса, у односу на остале језике тог времена. Међутим, у међувремену је на пољу програмских језика дошло до великог напретка, како у академским истраживањима, тако и у индустријској примени, те је Objective-C почео да показује знаке старости. Најважнија од тих слабости је управо наслеђе из C језика. Иако још увек изузетно важан, C језик је превише ниског нивоа у односу на савремене језике, те је изузетно лако, приликом програмирања, направити грешке и безбедносне пропусте, који у савременијим језицима нису допустиви (излажење из граница низа, цурење меморије или прерана деалокација итд). Друга мана Objective-C језика је необична синтакса. Ова синтакса, будући знатно другачија од уобичајене синтаксе у објектно орјентисаним језицима, може да збуни програмере навикнуте на друге језике и тиме их одбије од платформе. Поред тога, синтакса чини да многи конструкти морају бити непотребно дугачки, што може да смањи читљивост кода. Коначно, Objective-C је динамички језик и не поседује генеричке типове. Ово само по себи није мана, с обзиром на то да постоје бројни други модерни и успешни динамички језици, као што су Python и Ruby. Међутим, за компликованије системе, као што је су апликације на OS X платформи, статички језици, чији компајлери контролишу исправност типова пре извршавања, могу да буду јако значајни за спречавање великог броја грешака. С друге стране, мана многих статичких језика је то што је све типове неопходно експлицитно декларисати и означити. Језиком Swift изузетно добро се решавају сви побројани проблеми и уз то се додају нови језички конструкти, који омогућавају потпуно другачији начин програмирања, у односу на ObjectiveC.

Swift је потпуно нови језик, развијен од нуле, и, као такав, готово потпуно ослобођен C наслеђа. Али то не значи да два језика нису компатибилна. Напротив, већини кода, написаног у C језику, може се природно приступити из Swift-а. Поред тога, Swift има могућност коришћења било које библиотеке написане у Objective-C језику. Ово значи да су Cocoa и Cocoa Touch оквири потпуно доступни из Swift-а [5], те да се језик, од првог дана објављивања, може користити за развој комплетних iOS односно OS X апликација. Синтакса Swift-а је слична синтакси савремених језика као што су Java, C#, JavaScript итд, што значи да је позната и блиска већини програмера. Језик је статички,

компајлиран и користи аутоматско бројање референци за управљање меморијом, што му даје предности у перформансама у односу на језике који користе сакупљаче отпадака. Перформансе језика су врло блиске, а у неким случајевима и боље, од перформанси C језика и требало би још више да напредују, како језик сазрева. При дизајну језика велики нагласак је стављен на безбедност програмирања. И даље је могућ директан приступ меморији, коришћење показивача и слично, када је потребно програмирање ниског нивоа, нпр. за развој језгра оперативног система или драјвера за графичку карту. Ипак, овакав стил програмирања није стандардан и излази из оквира овог рада. Иако је језик строго статички, захваљујући алгоритму за аутоматско закључивање типова (Type Inference), типове је ретко потребно експлицитно обележавати. Није познато који се тачно алгоритам користи за закључивање типова, али претпостављамо да је у питању нека варијација Хиндли-Милнеровог алгоритма [6, 7]. Систем за закључивање типова, поред тога што олакшава писање кода, такође доприноси томе да се Swift може користити као скрипт језик. Swift поседује интерактивни REPL (Read–eval–print loop) и игралишта (Playgrounds), приказане на сликама 2 и 3, који омогућавају брз развој и тестирање у почетним фазама пројеката. Сви примери, дати у овом раду, произведени су коришћењем игралишта. Приликом коришћења интеактивне конзоле и игралишта, Swift кôд се и даље компајлира, само што се то чини у реалном времену.



Слика 2. Интерактивна Swift конзола (REPL). На слици је приказано дефинисање и позивање нове функције, као и приказивање резултата функције у реалном времену

Слика 3. Пример интерактивног Swift игралишта (Playground), преузет са званичног Swift блога [8]

Swift природно омогућава програмирање у оквиру неколико парадигми. Коришћењем слободних (глобалних) функција и глобалних променљивих, могуће је имплементирати чисто императивне програме, као у C језику. Поред тога, Swift поседује све конструкте и одлике неопходне за објектно орјентисано програмирање, као што су класе, наслеђивање, полиморфизам, интерфејси (протоколи) итд. Коначно, следећи савремене трендове, Swift омогућава и парадигму која је тема овог рада, функционално програмирање. Бројним конструктима, који омогућавају и олакшавају имплементирање ове парадигме, бавимо се у делу 2. Чини нам се да се тренутна Swift заједница грубо може поделити у две групе. У прву групу спадају програмери који имају искуства са Objective-C и осталим чисто објектно орјентисаним језицима. Чест је случај да такви програмери критикују Swift јер не успевају да одређене узорке, на које су навикли, елегантно имплементирају у Swift-у на исти начин. У другу групу спадају програмери који познају традиционалне функционалне језике, као што је Haskell. Ови програмери често директно преводе свој кôд из Haskell-а, користећи идентичну терминологију и истоимене операторе. Иако тај приступ, са чисто техничке стране, прилично добро успева Swift-у, барем две битне ствари му се могу приговорити. Прво, приступ је често потпуно непознат програмерима из прве групе, што је психолошка баријера ка прихватању функционалног програмирања. Друго, с обзиром на то да Swift није лењ језик, директно преведен Haskell кôд често показује перформансе слабије од оптималних. Главни циљ овог рада је да функционалном програмирању у Swift-у покуша да приђе са стране која би помирила ове две групе. Сматрамо да су многе идеје, преузете из језика као што су Haskell и F#, у које спадају чисте функције, кôд прилагођен композицији, примена функција вишег реда за апстракцију итд, изузетно корисне и тежимо да их усвојимо и применимо о Swift-у. Али, истовремено покушавамо да, кад год је то могуће, користимо синтаксу и терминологију

прилагођенију Swift-у и ближу објектно орјентисаним језицима, тако што користимо јасно именоване методе, уместо оператора, јаснije именоване функције (нпр. `foreach`, уместо `liftA`) итд. Све што је изложено и имплементирано у овом раду, односи се на тренутну верзију језика, Swift 1.2, која је објављена у марту 2015. године и још увек је у бета верзији. Имајући у виду да је Swift млад језик, који није још увек до краја завршен и који се изразито брзо развија, препоруке дате у овом раду би се у будућности могле променити. На пример, ако у овом раду препоручујемо одређени метод уместо опратора, а оператор исте намене буде укључен у неку будућу верзију језика, онда би свакако у будућности требало користити тај оператор. Рад нема амбицију, а ни могућност, да у том смислу донесе било какве трајне и опште важеће закључке, већ да допринесе обликовању функционалног стила програмирања у Swift-у, коришћењем рпилаза за који смо приметили да није толико заступљен у тренутно доступној литератури и интернет изворима.

2 Преглед функционалних одлика Swift језика

У овом одељку дајемо преглед оних својстава Swift-а, која су значајна за функционално програмирање. Нека од ових својстава су детљаније обрађена у наредним поглављима, која говоре о њиховој примени.

2.1 Функције као типови првог реда

Да би ламбда рачун био практично применљив у програмском језику, изузетно је важно да тај језик третира функције као прворазредне типове. Swift језик чини управо то, на следеће начине:

1) Константе и променљиве, како глобалне, тако и својства, могу бити типа функције:

```
let doSomething: Int -> Int struct SomeType {

let checkCondition: (String, String) -> Bool }
```

2) Функције су вредности и могу се, као такве, доделити константама и променљивама:

```
func square(number: Int) { return number * number }
let doSomething = square
```

3) Функције могу бити аргументи или враћене вредности других функција, чиме се детаљно бавимо у делу 3;

4) Функције имају своје литерале, тј. ламбда изразе, који се у Swift-у називају closures. На пример, претходни кôд се може другачије написати, коришћењем ламбда израза:

```
let doSomething: Int -> Int = { (number: Int) -> Int in return

   number * number

}
```

Ламбда изрази, који се састоје од једне линије кода, се готово увек могу краће написати. Компајлер може да закључи тип аргумента и враћене вредности, те је декларације типова могуће изоставити: `let doSomething: Int -> Int = { number in return number * number }` Кључну реч `return` није неопходно експлицитно наводити: `let doSomething: Int -> Int = { number in number * number }`

И, коначно, уместо експлицитних имена аргумената, могу се користити скраћена имена, `$0`, `$1` итд:

```
let doSomething: Int -> Int = { $0 * $0 }
```

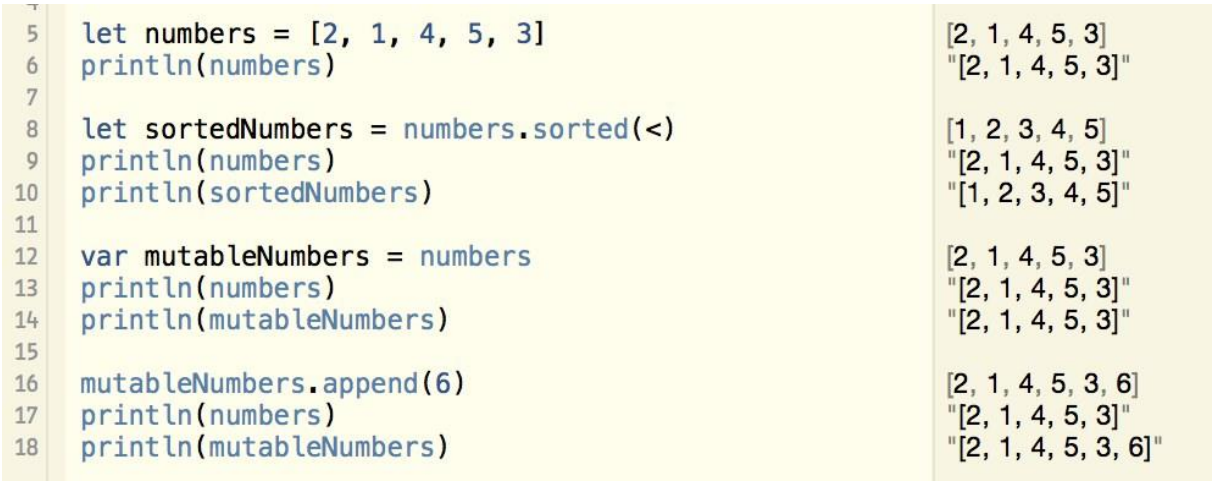
Када је ламбда израз последњи аргумент неке друге функције, може се користити погодна синтакса (Trailing Closure Syntax), па се тако `[1, 2, 3].map({ $0 * $0 })` може написати као `[1, 2, 3].map { $0 * $0 }`. Све ове синтаксичке погодности, за дефинисање ламбда израза, чине велики број функционалних идиома далеко

елегантнијим за имплементирање. Поред тога, побројане синтаксичке погодности олакшавају коришћење метода за имплементирање функционалних апстракција, у односу на друге језике који такође поседују методе, као што је F#. О парцијалној апликацији, још једној важној техници везаној за функције, говоримо детаљније у делу 3.3.

2.2 Структуре, као типови вредности

Важна одлика функционалног програмирања је референцијална транспарентност функција. Израз је референцијално транспарентан уколико се увек може заменити својом вредношћу, а да се при тој замени понашање програма не промени. У императивном програмирању изрази генерално нису референцијално транспарентни због споредних ефеката, те су технике, које се заснивају на комбиновању израза, знатно отежане. Пошто су изрази и њихово комбиновање суштински важни за функционално програмирање, пожељно је референцијалну транспарентност гарантовати на нивоу компајлера. У Haskell-у се ово осигурава тако што су све фунције чисте, тј. не могу имати споредне ефекте. То решење би било непрактично у језицима као што су Swift или F#, који подржавају више парадигми и морају да раде са оквирима и библиотекама које нису референцијално транспарентне. У Swift језику је ово решено тако што структуре и енумерације (уније дискриминатора) имају гарантовану семантику вредности, што значи да су или непроменљиве, или се при промени увек копирају. Swift не садржи примитивне типове. Цели бројеви, бројеви са покретним зарезом, низови, ниске итд. су имплементирани као структуре. Структуре су еквивалентне рекордима у језицима Haskell и F# и дефинишу се кључном речју `struct`, нпр.
`struct StructName {}`. Семантику вредности и понашање при промени је најлакше илустровати примером (Слика 4).

Вредност `numbers` на Слици 4 је низ. Пошто је вредност декларисана коришћењем кључне речи `let`, она се не може никако променити. Покушај да се то учини резултује пријавом грешке од стране компајлера. Сви методи и функције (нпр. `sorted`), које прихватају такав непроменљиви низ, враћају нови низ као резултат. Да би се низ могао променити (нпр. додавањем елемената), неопходно је обележити га кључном речју `var`, као што је учињено са вредношћу `mutableNumbers` на Слици 4. Важно је уочити следеће - иако је на почетку вредност `mutableNumbers` постављена да буде једнака вредности `numbers`, промена `mutableNumbers` не утиче на промену оригиналног `numbers` низа. То је управо одлика семантике вредности. Вредности се увек копирају, што није случај са референцама, на пример са инстанцама класа. Реч ,,увек" у овом случају треба схватити условно. Ако би заиста дошло до копирања приликом сваке доделе, перформансе би биле неприхватљиво слабе. Swift компајлер препознаје и копира вредности само ако заиста дође до промене. Постоји јавни API за ову оптимизацију, те се сви кориснички типови могу имплементирати тако да буду конзервативно копирани на овај начин, али детаљи те оптимизације превазилазе оквир овог рада. Још једно питање на које треба одговорити је како компајлер ,,зна" да метод `append` може



Слика 4. Илустрација семантике вредности. Низ `numbers` је константан и не може се променити. Сортирани низ (`sortedNumbers`) је копија почетног низа. Да би низ био променљив, мора бити декларисан са `var` (`mutableNumbers`), али и у том случају оригинални низ (`numbers`) остаје непромењен

бити примењен само на вредности декларисаној са `var`. Сви методи, који потенцијално мењају вредност структуре, морају бити обележени кључном речју `mutating`: `mutating func append(newElement: T)`

На овај начин компајлер може да забрани позивање таквих метода на константама, тј. `let` вредностима. Ово је изузетно битно за практично све што имплементирамо у овом раду. Знајући да се структуре са којима радимо не могу променити, можемо бити сигурни да ће функције у којима их користимо бити референцијално транспарентне, те да се могу компоновати или на други начин комбиновати, уланчавати итд. Наравно, понекад је копирање структура неприхватљиво са становишта перформанси. У тим случајевима је најбољи приступ да приватна имплементација функција буде императивна, а да јавни интерфејс буде функционално чист и референцијално транспарентан, као што смо показали у делу 5.

2.3 Тестирање подударања образаца (Pattern Matching)

Тестирање подударања образаца, за које смо у уводном делу рекли да је први пут уведено у језик ML, је важна конструкција за контролу тока у практично свим функционалним језицима. Конструкција је концептуално слична `switch` конструкцији из језика C и управо се кључна реч `switch` користи за тестирање подударања образаца у Swift-у. Међутим, док је традиционална `switch` конструкција углавном ограничена на буловске условне изразе и целе бројеве, иста конструкција у Swift-у је далеко моћнија. Општи облик конструкције је сличан синтакси C језика:

```
switch expression {
   case <pattern1>:
   case <pattern2>:
   ...
   default:
}
```

Прекид се подразумева након испуњеног услова, те наредба `break` није неопходна. Тип израза `expression` мора имплментирати оператор `~=`, да би се могао користити за тестирање подударања. Иако је могуће тај оператор имплементирати за било који кориснички тип, у овом делу приказујемо само она тестирања која су омогућена у стандардној библиотеци.

Најпре, `switch` се може користити са било којим типом који имплементира протокол `Equatable`:

```
func testName(name: String) { switch name { case "John", "Richard":
   println("The name is English.") case "Jovan", "Milan", "Janko":
   println("The name is Serbian.") default: println("Language of
   the name is not known.") }
}
```

Као што се може видети, више образаца, раздвојених зарезима, може се написати у оквиру једне `case` команде. Израз се може тестирати и на подударање са опсегом:

```
func testInteger(number: Int) { switch number { case
   1...10: println("The number is small.") case
   11...100: println("The number is medium.") case
   101...1000: println("The number is large.")
   default: println("The number is very large.") }
}
```

Коначно, израз се може тестирати на подударање са уређеним n-торкама, а то тестирање се може комбиновати са претходно описаним тестирањем са опсезима:

```
func testIntegers(number1: Int, number2: Int) { switch
   (number1, number2) { case (0, 0): println("Both are
   0.") case (0...10, _): println("Left number is
   small.") case (_, 0...10): println("Right number is
   small.") default: println("None of the numbers is
   small.") }
}
```

У претходном изразу је употребљен џокер знак (_), који се користи за игнорисање вредности. Некада је неопходно произвољну вредност, уместо игнорисања, везати за име и онда то име користити у даљим израчунавања. На пример:

```
func testIntegers(number1: Int, number2: Int) { switch (number1,
   number2) { case (0, 0): println("Both are 0.") case (0, let x):
   println("Left is zero, and square of the right is
      \(x * x).")
   case (let x, 0): println("Right is zero, and square of the left is
      \(x * x).") case
   (let x, let y):
      println("None is zero.") println("Square of
   the left is \(x * x).") println("Square of the
   right is \(y * y).") }
}
```

У претходном коду није неопходан `default` случај, јер су све могућности покривене трима опцијама. Веома важна примена везивања вредности за име, приликом тестирања подударања узорака, је за добијање вредности из унија

дискриминатора.Собзиромнавеликуважносттихтиповазафункционалнопрограмирање, о њима посебно говоримо у наредном делу.

2.4 Уније дискриминатора

Уније дискриминатора су веома важни типови у већини функционалних језика, укључујући Haskell и F#. Сличне су енумерацијама, али сваки случај може да садржи једну или више придружених вредности произвољног типа. Пример једне F# дисриминисане уније дат је у документацији језика [9]:

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Prism of width : float * float * height : float
```

У Swift-у се, за дефинисање еквивалентних типова, користи кључна реч `enum`:

```
enum Shape { case Rectangle(width : Double, length :
   Double) case Circle(radius : Double) case
   Prism(width : Double, Double, height : Double)
}
```

Ови типови су функционални еквивалент хијерархији класа у објектно орјентисаном програмирању. Тип `Shape` би могао бити дефинисан као апстрактна класа, а сваки од случајева као конкретна подкласа. Мана уније дискриминатора, у односу на хијерархију класа, је то што их је теже проширити, док је нову подкласу тривијално додати. С друге стране, при коришћењу унија дискриминатора, лакше је додати нову функционалност, то јест написати нови метод. Док би, у случају објектно орјентисаног решења, било неопходно дати метод имплементирати у свакој подкласи, у случају унија дискриминатора је довољан један метод у којем се подударање образаца користи за разликовање појединачних случајева и извлачење придружених вредности из њих:

```
extension Shape { func draw() { switch
   self { case let Rectangle(width,
   length):
          drawRectangle(width, length)
      case let Circle(radius):
      drawCircle(radius)
      case let Prism(width, length, height):
          drawPrism(width, length, height)
      }
   }
}
```

Уколико се неки од случајева уклони, или се нови случај дода, компајлер ће пријавити грешку. У језицима који подржавају обе врсте типова, поставља се питање избора између класа и унија дискриминатора. Класе су вероватно бољи избор за архитектуру великих система, које је касније неопходно надограђивати, док су уније дискриминатора бољи избор за моделовање објеката чија се хијерархија ређе проширује и мења (нпр. начини плаћања у систему за електронску трговину). Уније дискриминатора у Swift-у имају семантику вредности, те све, што је у том смислу написано за структуре, важи и за уније дискриминатора.

Уније дискриминатора су важне за дефинисање чистих функционалних структура података. На пример, листа је у F# језику дефинисана управо као унија дискриминатора:

```
type List<'T> =
     | ([]) of 'T list
     | (::) of Head: 'T * Tail: 'T list
```

Swift, у тренутној верзији, не дозвољава дефинисање рекурзивних типова, те је неопходно користити помоћни тип референце (класу), као прост омотач око рекурзивног типа. На овај начин је рекурзивну листу могуће дефинисати и у Swifty:

```
enum List<T> { case Empty(Box<List<T>>) case
     Cons(head: T, tail: Box<List<T>>)
}
```

Систематски опис функционалне листе и осталих функционалних структура података, изложио је Оказаки [10]. У овом раду смо унију дискриминатора користили за имплементацију `Result<T>` типа, у делу 4.4.3.

2.5 Генерички типови и протоколи

Иако је генеричко програмирање постало саставни и изузтено важан део главних објектно орјентисаних језика, као што су Java и C#, тек у последњој деценији 20. и првој деценији 21. века, овај тип програмирања је први пут уведен у функционални језик ML, давне 1972. године [11]. Генеричко програмирање је у функционалним језицима познато као параметарски полиморфизам. Swift омогућава дефинисање генеричких функција (нпр. `map<U>` и `bind<U>` описаних у делу 4), генеричких типова (нпр. `Array<T>` и `Optional<T>`) и генеричких протокола (нпр. `SequenceType`, коришћен у делу 5). За разлику од Haskell-а, Swift и F# не подржавају параметарски полиморфизам више врсте. Ово, поред осталог, значи да није могуће дефинисати генеричку монаду `M<T>`, где су и `M` и `T` генерички параметри. Могуће је, међутим, дефинисати појединачне монаде, код којих је само `T` тип генерички, а `M` конкретан, што и чинимо у делу 4.

2.6 Оператори

Оператори, строго гледано, нису неопходан део једног функционалног језика. Штавише у делу 2.7 предлажемо да се оператори користе само кад је заиста неопходно, те да се уместо њих радије бирају методи. Ипак, с обзиром на то да многи аутори користе операторе, преводећи Haskell кôд у Swift, у овом делу дајемо кратак опис њихове синтаксе у Swift-у. У овом раду смо операторе користили једино за имплементацију апликативног стила (део 4). Swift дозвољава реимплментацију (overloading) постојећих префиксних, инфиксних и суфиксних оператора. У том случају се оператори једноставно реимплементирају на потпуно исти начин као функције, с тим што је имплементације префиксних и постфиксних оператора неопхдоно обележити кључним речима `prefix` односно `sufix`. Поред тога, могуће је дефинисати и потпуно нове операторе, на следећи начин: `prefix|postfix|infix operator { associativity a precedence p }`

Аргументи унутар витичастих заграда, који одређују асоцијативност, односно приоритет оператора, могу се навести само за инфиксне операторе. Асоцијативност

може имати једну од три вредности, `left`, `right` или `none` (подразумевана вредност је `none`). Приоритет може бити било који број између 0 и 255, при чему је подразумевана вредност 100. Након што су нови оператори на овај начин декларисани, дефинишу се као било која друга функција, као што показујемо у делу 4, на примеру апликативног оператора.

2.7 Методи, глобалне функције и оператори

Важно техничко питање, које се поставља приликом имплементације функционалних библиотека у Swift језику, је избор између глобалних функција, оператора и метода. Глобалне функције нису добар избор за операције које су предвиђење за уланчавање јер захтевају угњеждене позиве (нпр. `bind(bind(monad, f1), f2)`, описана у делу 4). Помоћу оператора овај проблем се елегантно решава тако што се приликом позивања пишу између својих аргумената (нпр. `monad >>= f1 >>= f2`). Мана оператора је то што сами по себи нису довољно разумљиви и њихово значење се знатно разликује од језика до језика. Haskell програмер би одмах препознао `>>=` као оператор за монадско везивање, али истоимени оператор је у Swift стандардној библиотеци дефинисан као оператор за манипулисање битовима. Помоћу метода могу се решити оба проблема. Могу им се доделити разумљива имена, као глобалним функцијама, а позивају се веома слично као оператори (нпр. `monad.bind(f1).bind(f2)`). Swift омогућава дефинисање метода на свим типовима (класама, структурама, енумерацијма и унијама дискриминатора), а коришћењем екстензија методе је могуће додати чак и типовима за које није доступан изворни кôд. Коначно, потенцијално важна практична предност метода је то што су бољи за аутоматско комплетирање кода у развојним окружењима. Ово је важно не само за брже писање кода, већ и за спонтано откривање метода од стране програмера. Метод `bind` појавиће се у листи доступних метода сваки пут када програмер ради са датом монадом, што није случај са операторима. О доступности оператора може се сазнати једино кроз исчитавање документације или кроз интеракцију са другим програмерима. Идеално бисмо желели да све операције у овом раду имплементирамо као методе, али то није могуће због тренутних техничких ограничења језика. На пример, функција `flatten`, описана у делу 4, мора бити глобална јер тренутно није могуће специјализовати методе генеричких типова. Креатор Swift језика, Крис Латнер, наговестио је на званичном развојном форуму компаније Apple да би се ово могло променити у будућим верзијама компајлера [12] и његова изјава подупире нашу одлуку да изаберемо методе када год је то могуће.

3 Функције вишег реда

Функције вишег реда су оне функције које узимају друге функције као своје аргументе или враћају друге функције као резултат. Ове функције су централно својство функционалног пргорамирања. Апстракције, попут оних описаних у деловима 4 и 5, засноване су на функцијама вишег реда. У овом делу ћемо најпре описати важније функције вишег реда које су уграђене у стандардну библиотеку, а затим ћемо имплементирати неколико корисних функција које нису присутне у стандардној библиотеци, а инспирисане су сличним функцијама у F# језику.

3.1 Функције вишег реда уграђене у стандардну библиотеку

Вероватно најкорисније функције вишег реда су `map`, `filter` и `reduce`, које се користе за манипулацију секвенцама. Функција `map` као аргумент узима секвенцу и функцију трансформације. Функција трансформације се примењује на сваки елемент дате секвенце и нова секвенца се враћа као резултат. Функција `filter` узима секвенцу и функцију услова, а као резултат враћа нову секвенцу састављену од елемената улазне секвенце, који задовољавају дати услов. Функција `reduce` узима секвенцу, почетну вредност и функцију која комбинује тренутни елемент са тренутном вредношћу, а враћа крајњи резултат, чији је тип исти као тип прослеђене почетне вредности. Стандардна библиотека садржи два облика `map`, `filter` и `reduce` функција - као методе, дефинисане на типу `Array<T>`, и као глобалне функције, које као аргументе узимају било коју секвенцу. На Слици 5 илустрована је примена ових функција.

```
95  let numbers = Array(1...50)                                  [1, 2, 3, 4, 5, 6, 7,
96  let squares = numbers.map { $0 * $0 }                        (51 times)
97  let squaresDivisibleBy7 = squares.filter { $0 % 7 == 0 }     (51 times)
98  let sum = squaresDivisibleBy7.reduce(0, combine: +)          6860
```

Слика 5. Примена функције `map` за рачунање квадрата низа бројева, функције `filter` за филтрирање оних квадрата који су дељиви са 7 и функције `reduce` за рачунање суме филтрираних квадрата

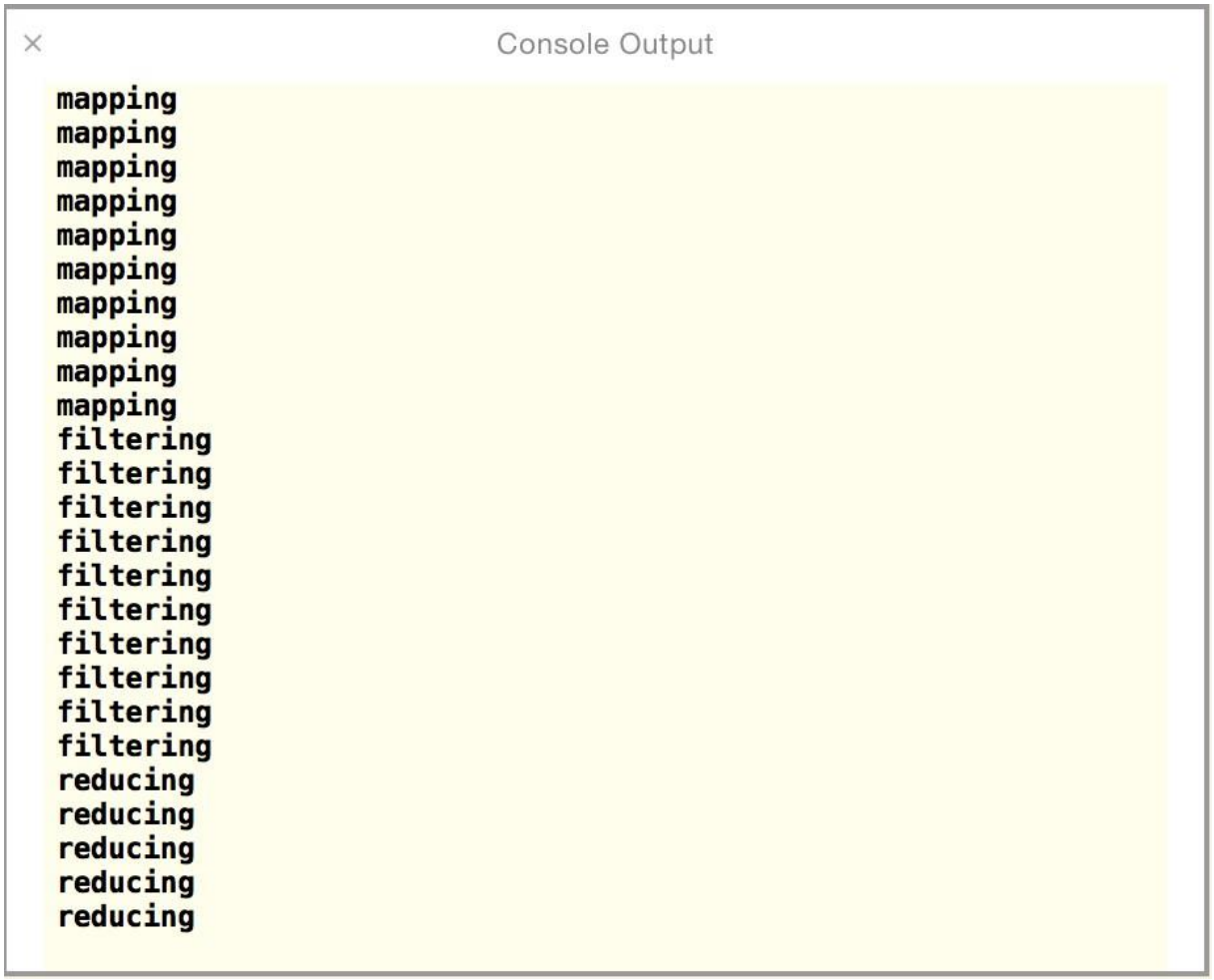
Када се `map` и `filter` користе уланчане, сваки међукорак доводи до копирања целих међурезултата, што, за велике улазе, доводи до значајног пада перформанси. За такве случајеве су, у стандардној библиотеци, дефинисане лење верзије ових функција, као методи на одговарајућим типовима. Да би се користиле лење верзије функција, неопходно је најпре дату секвенцу трансформисати у њен лењи еквивалент коришћењем функције `lazy`. Разлика између лењих и стриктних секвенци илустрована је сликама 6 и 7. У поглављу 5 предлажемо алтернативни приступ за решавање овог проблема.

3.2 Дефиниција и примена додатних функција вишег реда, присутних у F# језику

Језик F# садржи већи број корисних функција вишег реда за рад са листама, низовима и секвенцама. Те функције су дефинисане о одговарајућим модулима, `Seq`, `List` и `Array`. У овом делу описујемо како се најкорисније од тих функција могу имплементирати у Swift-у.

```
95  let numbers = Array(1...10)
96
97  let filteredSquares: [Int] = numbers
98      .map { println("mapping") ; return $0 * $0 }
99      .filter { println("filtering") ; return $0 % 2 == 0 }
100
101 let sum = filteredSquares.reduce(0) {
102     println("reducing") ; return $0 + $1
103 }
```

(a)



(b)

Слика 6. (a) Кôд приказан на слици је функционално идентичан коду са слике 5, осим што је модификован да штампа информацију о операцији која се тренутно обавља; (b) Одштампани излаз кода са слике (a). Може се видети да свака од три функције узрокује поновни пролазак кроз низ. Ово може да резултује смањеним перформансама за велики улазни низ

```
107 let lazyFilteredSquares = lazy(numbers)
108     .map { number -> Int in println("mapping") ; return number * number }
109     .filter { number -> Bool in println("filtering") ; return number % 2 == 0 }
110
111 let lazySum = reduce(lazyFilteredSquares, 0) {
112     println("reducing") ; return $0 + $1
113 }
```

(a)

(b)

Слика 7. (a) Кôд је сличан коду са слике 6.a, осим што је модификован да користи лење секвенце; (b) Одштампани излаз кода са слике (a). Овога пута се кроз низ пролази само једном. Функције `map` и `filter`, дефинисане на лењим секвенцама, не узрокују пролазак кроз низ. Кроз низ се пролази тек када је резултат заиста потребан, тј. након позивања функције `reduce`

Функције `map2` и `map3` су сличне функцији `map`, али се могу користити са две односно три листе. Трансформациона функција узима, редом, по један аргумент из сваке листе. Уколико примљене листе нису исте дужине, функције у F# језику пријављују изузетак. Уместо тога, верзије које имплементирамо у овом раду једноставно престају да трансформишу и враћају резултат када се краћа од две листе истроши. С обзиром на то да је у Swift-у дозвољено користити различита имена за функције са различитим бројем и типом аргумената, суфикс није неопходан. Све функције се могу једноставно назвати `map`. Функције имплементирамо тако да се могу применити на било коју секвенцу (`SequenceType`), на следећи начин:

1) `map2`:

```
func map<
   S1: SequenceType,
   S2: SequenceType,
   U>( s1: S1, s2:
   S2, transform: (
   S1.Generator.Element,
```

```
    S2.Generator.Element) -> U)

    -> [U] {

    var s1Generator = s1.generate() var
    s2Generator = s2.generate() var result =
    [U]() while let nextS1 = s1Generator.next(),
    let nextS2 = s2Generator.next() {
    result.append(transform(nextS1, nextS2))
    } return
    result
}
```

2) `map3`:

```
func map<
    S1: SequenceType,
    S2: SequenceType,
    S3: SequenceType, U>(
    s1: S1, s2: S2, s3:
    S3, transform: (
    S1.Generator.Element,
    S2.Generator.Element,

    S3.Generator.Element) -> U)

     > [U] {
    var s1Generator = s1.generate()
       var s2Generator = s2.generate() var s3Generator =
       s3.generate() var result = [U]() while let nextS1 =
       s1Generator.next(), let nextS2 = s2Generator.next(),
       let nextS3 = s3Generator.next() {
       result.append(transform(nextS1, nextS2, nextS3))
       } return
       result
}
```

На Слици 8, приказана је примена ове две функције.

```
let names = ["Albert", "Erwin", "Richard"]
let surnames = ["Einstein", "Schrödinger", "Feynman"]
let scientists = map(names, surnames) { $0 + $1 }
println(scientists)
```

```
["Albert", "Erwin", "Richard"]
["Einstein", "Schrödinger", "Feynman"]
(4 times)
"[AlbertEinstein, ErwinSchrödinger, RichardFeynman]"
```

(a)

```
let results = map([1.0, 2.2, 3.0], [4, 5, 6], [7, 8, 9]) {
    pow($1 + $2, $0)
}
println(results)
```

```
[11, 282.2769232445831, 3375]
(3 times)

"[11.0, 282.276923244583, 3375.0]"
```

(b)

Слика 8. (a) Примена `map2` функције за конструисање низа пуних имена из низа имена и низа презимена; (b) Примена `map3` функције за израчунавање вредности израза, у којем се елементи другог и трећег низа сабирају, па потом подижу на степен дефинисан првим низом

Функција `forall` враћа вредност `true`, ако сваки елемент листе задовољава дати услов, или `false`, у супротном случају. Имплементирамо је као екестензију `Array<T>` типа, коришћењем метода `reduce`:

```
extension Array { func forAll(predicate: T ->
   Bool) -> Bool { return reduce(true) { result,
   element in result && predicate(element)
      }
   }
}
```

Сличан овом методу је метод `forAny`, који враћа `true` ако било који елемент из листе задовољава дати услов:

```
extension Array { func forAny(predicate: T ->
   Bool) -> Bool { return reduce(false) {
   result, element in result ||
   predicate(element)
      }
   }
}
```

На Слици 9 илустрована је примена `forAll` и `forAny` метода.

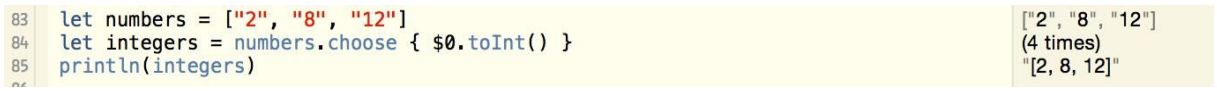
```
let test1 = [1, 2, 3].forAll { $0 < 4 }                                          (4 times)
println(test1)                                                                   "true"
let test2 = [1, 2, 3].forAll { $0 > 2 }                                          (2 times)
println(test2)                                                                   "false"
let test3 = Array(stride(from: 3, to: 103, by: 3)).forAll { $0 % 3 == 0 }        (35 times)
println(test3)                                                                   "true"
let test4 = [1, 2, 3].forAny { $0 == 2 }                                         (3 times)
println(test4)                                                                   "true"
let test5 = [1, 2, 3].forAny { $0 == 4 }                                         (4 times)
println(test5)                                                                   "false"
```

Слика 9. Примена `forAll` метода за проверу услова да су сви елементи низа мањи од 4, већи од 2, односно дељиви са 3 и примена `forAny` метода за проверу услова да је било који елемент низа једнак 2, односно једнак 4

Функција `choose` примењује дату функцију на сваки елемент и, уколико функција врати резултат `Some`, тај резултат се укључује у нову листу, која се враћа као крајњи резултат:

```
extension Array { func choose<U>(selector: T ->
   U?) -> [U] { var result = [U]() for element
   in self { if let value = selector(element) {
   result.append(value)
         } } return
      result
   }
}
```

На Слици 10 приказана је примена овог метода, а у делу 5 приказано је генералније решење сличних проблема.

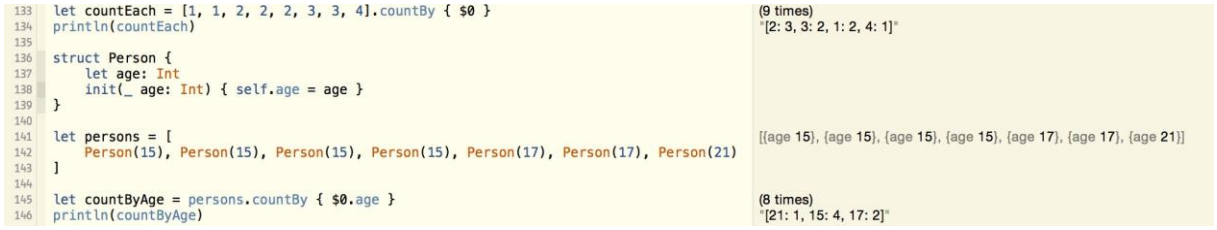


Слика 10. Примена метода `choose` за трансформацију низа ниски у низ целих бројева

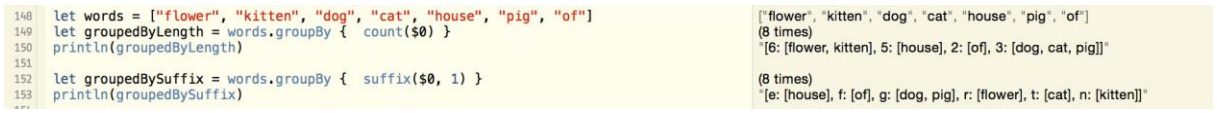
Две веома корисне F# функције из `Seq` модула су `countBy` и `groupBy`, које се користе за бројање, односно груписање елемената секвенце по датом кључу. Функције се могу користити за моделовање хистограма и сличних графикона, као и за разне друге статистичке анализе. У F# језику ове две функције враћају секвенцу кључева и вредности за те кључеве. Одговарајућа секвенца у Swift-у језику је `Dictionary<T>` из стандардне библиотеке, те њу користимо за имплементацију ове две функције, на следећи начин:

```
extension Array { func countBy<Key: Hashable>(counter: T -> Key) ->
   [Key: Int] { var result = [Key: Int]() for element in self { let
   currentKey = counter(element) let currentCount =
   result[currentKey] result[currentKey] = currentCount.map { $0 +
   1 } ?? 1
      } return
      result
   }
}

extension Array { func groupBy<Key: Hashable>(grouper: T -> Key) ->
   [Key: [T]] { var result = [Key: [T]]() for element in self { let
   currentKey = grouper(element) let currentGroup =
   result[currentKey] result[currentKey] = currentGroup.map { $0 +
   [element] } ?? [element]
      } return
      result
   }
}
```

На Слици 11 приказано је неколико примена ових важних метода.



(a)



(b)

Слика 11. (а) Примена `countBy` метода за просто пребројавање елемената низа, односно за пребројавање особа по годинама старости; (b) Примена `groupBy` метода за груписање речи по дужини, односно по заједничком последњем слову

Функција `partition` дели секвенцу на подсеквенцу елемената који задовољавајудатиусловиподсеквенцуелеменатакојинезадовољавају.Имплементирамо је као метода на `Array<T>` типу, на следећи начин:

```
extension Array { func partition(predicate: T -> Bool) -> (passed:
   [T], failed:

      [T]) { var passed = [T]() var failed = [T]() for
      element in self { if predicate(element) {
      passed.append(element) } else {
      failed.append(element) }
      } return (passed,
      failed)
   }
}
```

```
let divisibleByThree = Array(1...50).partition { $0 % 3 == 0 }
println(divisibleByThree.passed)
println(divisibleByThree.failed)
```

```
(51 times)
"[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]"
"[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28
```

Слика 12. Примена метода `partition` за поделу низа целих бројева у подниз бројева дељивих са три и подниз осталих бројева

Сликом 12 илуструјемо примену управо дефинисаног метода.

Изузетно важна функција је из `Seq` модула је `collect`, која је синоним за монадску `bind` операцију. Ову функцију имплементирамо и детаљно разматрамо у делу 4.

3.3 Парцијална апликација

Парцијална апликација је примена функције на део њених аргумената, при чему се добија нова функција која узима остале аргументе. Функције које дозвољавају парцијалну апликацију се називају Каријеве функције, по математичару Хаскелу Карију, по коме и језик Haskell носи име. Тај облик је подразумеван у Haskell-у и F#-у, у којима је свака функција заправо функција једног аргумента. Ово илуструјемо (користећи F# синтаксу), једноставном `add` функцијом, која сабира два броја:

```
let add x y = x + y
```

Функција се, уобичајено, може позвати са два аргумента и вратити њихов збир као резултат. На пример, резултат од `add 2 5` је 7, као што је и очекивано. Међутим, функцију је такође могуће позвати само са једним аргументом. У том случају је резултат нова функција, која узима један број и додаје га оном на којем је функција `add` првобитно позвана: `let addTwo = add 2`

Новафункцијасеможекориститикаобилокојадруга,тојерезултатод`addTwo 8` једнак 10. Парцијална апликација није много занимљива ни корисна у овако једноставним случајевима, али представља моћан алат за конструкцију специјализованих функција из генералних функција вишег реда. Функција `fold` из F# је слична претходно описаној функцији `reduce` из Swift језика. Основна разлика је то што су секвенца, листа или низ, на које се функција примењује, наведени као последњи аргумент. Ово омогућава да се, изузетно елегантно, из функције `fold` изведу специјализоване функције за суму, производ итд, на следећи начин:

```
let sum = List.fold (+) 0 let
product = List.fold (*) 1
```

Парцијална апликација је могућа и у Swift-у, али се функције морају написати на посебан начин. Да бисмо то показали, имплементирамо функцију `fold` из F#:

```
func fold<T, U>(initial: U)(combine: (T, U) -> U)(array: [T]) -> U {

   return array.reduce(initial, combine)

}
```

Овако дефинисана, функција `fold` се може парцијално применити и користи за конструкцију специјализованих функција на исти начин као у F# језику:

```
let sum = fold(0)(+) let
product = fold(1)(*)
```

За крајњег корисника, функције `sum` и `product` се користе исто као да су дефинисане стандардним путем, на пример `sum([1, 4, 5])`.

С обзиром на то да ниједна функција у стандардној библиотеци није написана у облику погодном за парцијалну апликацију и да су многе од њих дефинисане као методи на одговарајућим типовима, поставља се питање да ли је те методе могуће искористити на исти начин као претходно дефинисану `fold` функцију. Одговор је потврдан. Методи се могу парцијално применити, тако што се њихово позивање упакује у ламбда израз. Претходно дефинисане функције `sum` и `product` могу се, користећи метод `reduce`, имплементирати на следећи начин:

```
let sum: [Int] -> Int = { $0.reduce(0, +) } let

product: [Int] -> Int = { $0.reduce(1, *) }
```

Ако се изузме потреба за експлицитним навођењем типа функције, ова дефиниција је подједнако елегантна као претходна која користи глобалну, парцијално аплицирану, `fold` функцију. Сматрамо да је та разлика незнатна у свакодневном раду , те и у овом случају предност дајемо методима.

3.4 Композиција функција

Композиција је кључна техника за практично сваку апстракцију у функционалним језицима. Строго значење ове речи није прецизно дефинисано. У најширем смислу, композиција је начин да се, од мањих и простијих компонената (најчешће функција), изграде сложеније компоненете и системи. У ужем смислу, под композицијом се најчешће подразумева математичко значење, то јест операција при којој се од две функције добија трећа, чији је резултат једнак резултату појединачне уланчане примене прве две функције:

$$(f \circ g)(x) = f(g(x))$$

У језику Haskell, оператор за композицију (.) дефинисан је на исти начин:

```
f . g = \ x -> f (g x)
```

У језику F#, дефинисан је оператор са сличним значењем, >>. Једина разлика, у односу на математички и Haskell оператор за композицију, је редослед примене функција. Оператор >> се прво примењује на свој леви аргумент, па онда резултат операције прослеђује десном аргументу:

```
let (>>) f g = fun x -> g (f x)
```

Стандардна библиотека језика Swift не садржи оператор за композицију, али је исти тривијално дефинисати. Неки аутори [13] предложили су имплементацију сличну оној из F# језика уз коришћење >>> оператора, пошто је >> резервисан као оператор за померање битова:

```
func >>> <T, U, V>(f: T -> U, g: U -> V) -> T -> V {
   return { g(f($0)) }
}
```

С обзиром на то да овај оператор није уграђен у језик и да је непознат програмерима који немају исксутва са другим функционалним језицима, у наредним деловима разматрамо алтернативне начине композиције и комбиновања функција.

3.5 Алтернативни приступи композицији у Swift језику

Ајдхоф и коаутори [13] су имплементерали већину функционалних техника користећи глобалне функције и оператор >>> за композицију, што је, мање или више, директан превод одговарајућег Haskell кода. Ови аутори користе структуре као омотаче функција само у случају генеричких функција, за које није могуће дефинисати синониме (typealias). У овом делу покушавамо да покажемо да су структуре, заправо, добар начин за дизајнирање свих функција које је потребно компоновати на неки начин. Такође показујемо како се израчуната својства (Computed Properties) могу користити као алтернатива композицији. Овим не тврдимо да су решења, која предлажемо, супериорнија од оних директно инспирисаних Haskell језиком. Напротив, сматрамо да је оператор >>> елегантно решење за композицију, али, као што је случај и са многим другим операторима, значење овог оператора није одмах очигледно програмерима који немају претходног исксутва са F# или Haskell језицима. Сматрамо да су методи и својства одличан алтернативни начин исказивања бројних функционалних техника у Swift-у, с обзиром на то да захтевају само мало више кода од традиционалних начина конпозиције, а при томе не уводе ништа ново што би програмери, који познају Swift језик, морали да науче.

3.5.1 Коришћење метода и израчунатих својстава за композицију

CoreImage је библиотека из Cocoa и CocoaTouch оквира, која се користи за манипулисање сликама. Библиотека је изузетно моћна, али је написана у ObjectiveC језику, те није увек најпогоднија за коришћење из Swift-а. Да би се то превазишло. Ајдхоф и коаутори су предложили следеће решење:

1) Тип `Filter` је дефинисан као синоним за функцију `CIImage -> CIImage`:

```
typealias Filter = CIImage -> CIImage
```

2) За сваки жељени `Filter` дефинисана је функција облика `func filter(parameters...) -> Filter`

3) Претходнодефинисани`>>>` операторкоришћењезакомпозицијупростихфилтера у сложеније, на следећи начин:

```
let compoundFilter = filter1(parameters...) >>>
   filter2(parameters...) ...
```

Детаљи њихове имплементације могу се прочитати у [13]. Алтернативно решење, вероватно ближе (тренутном) духу Swift језика, је да се филтери дефинишу као методи, односно израчуната својства, на самом `CIImage` типу:

```
extension CIImage { func blur(radius:
   Double) -> CIImage { let parameters:
   Parameters = [ kCIInputRadiusKey:
   radius, kCIInputImageKey: self
      ] let filter = CIFilter(name: "CIGaussianBlur",
      parameters:
         parameters)
      return filter.outputImage
   }

   func compositeOver(overlay: CIImage) -> CIImage {
      let parameters: Parameters = [
      kCIInputBackgroundImageKey: self,
      kCIInputImageKey: overlay
      ] let filter = CIFilter( name:
      "CISourceOverCompositing",
      parameters: parameters
      ) let cropRect = extent() return
      filter.outputImage.imageByCroppingToRect(cropRect)
   }

   func overlayWithColor(color: NSColor) -> CIImage { let
      colorImage = CIImage(color: CIColor(color: color))
      return compositeOver(colorImage)
   }
}
```

Претходно дефинисани филтери се могу користити за дефинисање сложенијих филтера, кроз нове методе или кроз израчуната својства, ако сложени филтери не узимају аргументе:

```
extension CIImage { var bluredAndRed: CIImage { return
   blur(blurRadius).overlayWithColor(overlayColor) }
}
```

Примена дефинисаног `bluredAndRed` сложеног филтера за замућење и бојење фотографије у jpeg формату, илустрована је на Слици 13. У случају када су прости филтери дефинисани као функције, а сложени филтер као композиција тих функција, еквивалентни кôд се може написати на следећи начин:

```
let blurAndRed = blur(blurRadius) >>> overlayWithColor(overlayColor) let
bluredAndRedBird = burAndRed(birdImage)
```

Овај кôд је тек нешто краћи од кода који користи израчуната својства, али је такође мање разумљив типичном Swift програмеру. Чак и ако игноришемо неразумљивост, израчуната својства имају предност над глобалним функцијама у другим контекстима, у којима композиција није примарни захтев. Некада, међутим,

```
51  extension CIImage {
52      var bluredAndRed: CIImage {
53          return blur(blurRadius).overlayWithColor(overlayColor)
54      }
55  }
56
57  let birdImage = CIImage(data: NSImage(named: "bird.jpg")?.TIFFRepresentation)
58  let bluredAndRedBird = birdImage.bluredAndRed
```

Слика 13. Примена сложеног филтера, дефинисаног као израчунато својство на типу `CIImage`

није семантички погодно дефинисати израчунато својство или метод на датом типу. У таквим случајевима је функције које се компонују могуће написати као својства структуре, уместо као глобалне функције. У наредном делу ћемо показати како је такве структуре могуће компоновати, коришћењем метода, уместо оператора.

3.5.2 Коришћење `Function<A, B>` структуре за композицију

структурунујемоУместо да директно користимо функцију општег типа`applyFunction<A, B>`јер је у питању функција која се примењује на аргумент типа, која садржи својство типа `A ->AB`¯. Ово својство име-`> B`, дефинишемо`A`:

```
struct Function<A, B> {
    let apply: A -> B


    init(_ apply: A -> B) { self.apply
        = apply
    }
}
```

На овој структури се може написати метод чије је значење аналогно значењу `>>>` оператора. За разлику од оператора, методима се може дати описније име, на пример `followedBy`:

```
extension Function { func followedBy<C>(anotherFunction: Function<B,
    C>) -> Function<A, C> { return Function<A, C> {
    anotherFunction.apply(self.apply($0)) }
    }
}
```

На Слици 14 приказана је примена ове структуре за композицију.

```
157  let toInt = Function<String, Int?> { $0.toInt() }                    (3 times)
158  let square = Function<Int?, Int?> { $0.map { $0 * $0 } }              (4 times)
159  let toIntAndSquare = toInt.followedBy(square)                         {(Function)}
160  toIntAndSquare.apply("56")                                            3136
161  toIntAndSquare.apply("A")                                             nil
```

Слика 14. Примена `Function<A, B>` структуре за композицију функција

Овом приступу би се са правом могло приговорити да је у питању корак уназад. Функције у Swift језику су типови првог реда, равноправни са свим осталим типовима. Једна од основних предности функција, као типова првог реда, је то што нису потребни додатни типови да их моделују, попут делегата у C# језику. Овај приговор је свакако значајан за случај када се функције користе као аргументи функција вишег реда. На пример, `let summed = [1, 2, 3].map { $0 * $0 }`

је свакако краће, елегантније и читљивије од `let summed = [1,`

`2, 3].map(Function { $0 * $0 }.apply)`

Дужа верзија, која користи `Function<A, B>` структуру, нема ниједну предност, само уводи додатни синтаксички шум. Наравно, могуће је за нијансу упростити другу верзију, тако што би се написао метод `map`, који као аргумент узима `Function<A, B>`, уместо `A -> B`. Али и даље се јавља више кода него што је неопходно у Swift језику, а

истовремено нема других предности. С друге стране, када се `Function<A, B>` користи за писање нових функција, намењених за композицију, додатни синтаксички шум је знатно мањи, а читљивост кода, по нашем миљењу, нешто већа. Композиција приказана на Слици 14 би се, без коришћења `Function<A, B>` структуре, а уз коришћење `>>>` оператора, могла написати на следећи начин:

```
let toInt: String -> Int? = { $0.toInt() } let
square Int? -> Int? = { $0.map { $0 * $0 } } let
toIntAndSquare = toInt >>> square
toIntAndSquare("56") toIntAndSquare("A")
```

За разлику од коришћења метода и израчунатих својстава за композицију (део 3.5.1), што је говото увек боље од коришћења `>>>` оператора, коришћење структуре `Function<A, B>` не можемо генерално да препоручимо као боље решење. Уколико Swift стандардна библиотека у наредним верзијама добије оператор за композицију, то би био дефинитивни знак да је коришћење тог оператора и глобалних функција пожељније него коришћење `Function<A, B>` или сличне структуре.

Композиција није једини начин комбиновања функција. Чест је случај да се две функције комбинују у нову функцију, која примењује обе функције на дати аргумент, а потом комбинује добијене резултате (што је различито од уланчавања). За такве случајеве структуре имају знатно очигледнују предност у односу на просту математичку композицију, што ћемо показати у следећем делу.

3.5.3 Коришћење произвољних структура за произвољна комбиновања функција

Погодан пример за илустрацију ове врсте комбиновања је функција помоћу које се одређује да ли дата тачка припада одређеном региону простора. Ајдхоф и сарадници су предложили такву функцију као део симулације (или игре) борбе бродова, инспирисани сличним Haskell решењем [13]. Функција има тип `Point -> Bool` и додељен јој је алијас `Region`: `typealias Region = Point -> Bool`

Сваки елементарни регион је дефинисан малом функцијом, на пример:

```
func circle(radius: Distance) -> Region { return { point in
  sqrt(point.x * point.x + point.y * point.y) <= radius }
}
```

Поред тога, дефинисан је низ функција које трансформишу елементарне регионе:

```
func shift(offset: Point, region: Region) -> Region { return { point
   in let shiftedPoint = Point(x: point.x - offset.x, y: point.y -
   offset.y)
      return region(shiftedPoint)
   }
}
```

```
func invert(region: Region) -> Region { return
   { point in !region(point) }
}
```

Други могући начин решавања овог проблема је коришћење приступа описаног у делу 3.5.1. Наиме, уместо дефинисања глобалне `Region` функције, за сваки елементарни регион се може дефинисати метод на самом `Point` типу, који одређује да ли инстанца тог типа припада региону. Тако би се кружни регион могао дефинисати на следећи начин:

```
extension Point { func isInCircle(radius: Distance) -> Region { return
   sqrt(point.x * point.x + point.y * point.y) <= radius }
}
```

Уместо да `shift` трансформише глобалну `Region` функцију, могу се директно трансформисати саме иснтанце `Point` типа:

```
extension Point { func shifted(offset: Point) -> Point { return
   Point(x: point.x – offset.x, y: point.y – offset.y) }
}
```

Овај прилаз има барем два недостатка. Први је семантички. Наиме, док региони и трансформације попут `shift` природно припадају `Point` типу, трансформације попут `invert` би морале да се дефинишу на `Bool` типу, а то свакако не би имало смисла. Други недостатак је техничке природе. Поред трансформација које примају један регион, а враћају други (као што је `shift`), постоји велики број трансформација које два региона комбинују у један. На пример:

```
func intersection(region1: Region, region2: Region) -> Region { return
   { point in region1(point) && region2(point) }
}

func difference(region: Region, minusRegion: Region) -> Region { return
   intersection(region, invert(minusRegion))
}
```

Овакве трансформације је тешко имплементирати користећи методе и својства на типу `Point`. Решење које предлажемо је слично решењу за композицију, кога смо изложили у делу 3.5.2. Уместо да `Region` буде типа функције `Point -> Bool`, имплементирамо тај тип као структуру, чије је својство функција типа `Point -> Bool`:

```
struct Region { let containsPoint:
   Point -> Bool
```

```
    init(_ containsPoint: Point -> Bool) {
        self.containsPoint = containsPoint
    }
}
```

Након што је тип дефинисан, све претходно описане трансформације се могу написати као методи или својства на овом типу:

```
extension Region { func shift(offset:
    Point) -> Region { return Region {
    point in let shiftedPoint = Point( x:
    point.x – offset.x, y: point.y –
    offset.y
          ) return
    self.containsPoint(shiftedPoint) } }

    var inverted: Region {
        return Region { point in
            !self.containsPoint(point)
        }
    }

    func intersectWith(anotherRegion: Region) -> Region { return
        Region { point in
            self.containsPoint(point) &&
    anotherRegion.containsPoint(point) } }

    func differenceFrom(anotherRegion: Region) -> Region {
    return self.intersectWith(anotherRegion.inverted) }
}
```

Ајдхоф и сарадници илуструју примену својих функција на следећем примеру:

```
func inRange( ownPosition:
    Position, target: Position,
    friendly: Position, range:
    Distance) -> Bool {

    let rangeRegion = difference( circle(range),
                          circle(minimumDistance) )
    let targetRegion = shift(ownPosition, rangeRegion) let
    friendlyRegion = shift(friendly, circle(minimumDistance)) let
    resultRegion = difference(targetRegion, friendlyRegion) return
    resultRegion(target)
}
```

Дефиниција исте функције, коришћењем `Region` структуре, приказана је на слици 15.

```
func inRange(
    ownPosition: Point,
    target: Point,
    friendly: Point,
    range: Distance
    ) -> Bool {

    let minimumDistanceRegion = circle(minimumDistance)                          {(Function)}
    let rangeRegion = circle(range).differenceFrom(minimumDistanceRegion)        {(Function)}
    let targetRegion = rangeRegion.shift(ownPosition)                            {(Function)}
    let friendlyregion = minimumDistanceRegion.shift(friendly)                   {(Function)}
    let resultRegion = targetRegion.differenceFrom(friendlyregion)               {(Function)}
    return resultRegion.containsPoint(target)                                    false
}

let currentPosition = Point(x: 17.5, y: 24.3)                                    {x 17.5 y 24.3}
let targetPosition = Point(x: 84.0, y: 77.2)                                     {x 84 y 77.2}
let friendPosition = Point(x: 185.4, y: 233.3)                                   {x 185.4 y 233.3}
let range = Distance(250.0)                                                      250

if inRange(currentPosition, targetPosition, friendPosition, range) {
    println("Shoot!")
} else {
    println("Don't shoot, you'll miss!")                                         "Don't shoot, you'll miss!"
}
```

Слика 15. Примена `Region` структуре за одлучивање о нападу на противничке бродове. У конкретном случају противник није у безбедном опсегу, те је боље штедети муницију

За разлику од примене `Function<A, B>` структуре за композицију функција, сматрамо да структура `Region` има очигледне предности у односу на глобалну `Point -> Bool` функцију и евентуалне операторе. Додатна предност овог приступа је то што ће се природно уклопити у екстензије протокола када једном буду уведене у језик, као што је наговестио Латнер [12].

4 Функтори, монаде и апликативни функтори

Монаде су апстрактне структуре настале у оквиру теорије категорија. Значај монада за програмирање описао је Мођи у свом раду [14], а у практичну употребу су ушле најпре кроз програмски језик Haskell [15], [16], [17]. Будући чист функционални језик, Haskell користи монаде као једини начин за имплементацију споредних ефеката и то је била њихова првобитна намена. Ипак, монаде су знатно генералније структуре и у најширем смислу моделују операције које је могуће уланчавати и као такве су потенцијално корисне и у другим језицима, који нису фунционално чисти. Све до недавно, скоро да нису коришћене ван Haskell језика, што због своје опскурне природе, што због чињенице да синтакса, типови и семантика већине језика, нарочито оних најзаступљенијих, као што су C++, Java, Objective-C, C# итд, нису најпогоднije за имплементацију монадског кода. Последњих година се та ситуација мења, те се монаде, у разним облицима, појављују у језицима који се користе на главним платформама, као што су Scala (Java платформа) [18], и F# (.NET платформа) [19]. У овом делу показујемо како се концепт монаде може искористити за побољшавање свакодневног рада са неколико најкоришћенијих типова у Swift језику.

4.1 Дефиниција

Појмови функтора и монаде прецизно и формално су дефинисани у теорији категорија [жује објекат20F](. Ако су C и D категорије, тада функтор F свакомX) $\in^{)}{}_{B}{}^{B}$и сваком морфизму. Сваки функтор мора да очува морфизме идентитета и$f : X \to Y \in A$ придружује морфизам$X \in A$ придру-

$F(f) : F(X) \to F(Y$

композиције. Ендофунктор је функтор који дефинише пресликавање категорије у себе саму. Монада је ендофунктор $F : C \to C$, заједно са две придружене$C,$ и природне операције,μцијативност.: $F \circ \ F \to F$. Свака монада мора да задовољава леви и десни идентитет и асо-$\eta : I \to F$, где је I индентитет фунтор у категорији

4.2 Имплементација у програмском језику Haskell

Haskell имплементира функтор и монаду као класе типова [21], [22].

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b

class Monad m where
```

```
(>>=) :: m a -> (a -> m b) -> m b

(>>) :: m a -> m b -> m b return ::

a -> m a fail :: String -> m a
```

Функција `fmap` у функтор класи дефинише пресликавање датог фунтора у себе самог. Класа монада је на први поглед другачија од дефиниције монаде из теорије категорија. Природна операција η је присутна у облику функције `return`, која узима објекат датог типа и поставља га у контекст монаде. Међутим, операција μ није декларисана у класи. Додатно, монада не наслеђује класу `Functor`.

Уместо тога, у класи се дефинише `>>=` оператор, који је еквивалентан секвенци операција `fmap` и μ, а практично је далеко кориснији од појединаче примене сваке од ове две операције, јер омогућава уланчавање монадских израза. Специјалну улогу за рад са монадама има `do`-конструкција која се заснива на овом оператору. Појединачне операције су присутне, уколико су потребне, у облику функција `liftM` односно `join`. Разлог због којег монада тренутно не наслеђује класу `Functor` је историјске природе и та ситуација би требало да се промени у новој итерацији компајлера, најављеној за фебруар 2015. године. Оператор `>>` је сличан оператору `>>=`, али игнорише резултат свог левог аргумента и узима у обзир само његов монадски ефекат. Ако монада моделује операцију, чији резултат може бити вредност или грешка, `>>=` извлачи евентуални успешни резултат из свог левог аргумента и прослеђује га десном аргументу, који је функција. Оператором `>>` се такође проверава да ли његов леви аргумент садржи успешни резултат или грешку, али у случају успеха не користи дати резултат, већ враћа свој десни аргумент, који је монада, а не функција.

МекБрајд и Патерсон су показали да је монадска апстракција често сувишна у случајевима у којима не постоји међузависност између појединачних корака монадског ланца [23] и за такве случаје су предложили апстракцију апликативног функтора, која је у језику Haskell такође имплементирана као класа типова [24]:

```
class (Functor f) => Applicative f where pure
   :: a -> f a

   (<*>) :: f (a -> b) -> f a -> f b
```

Функција `pure` је идентична монадској функцији `return`. Оператор `<*>` омогућава да се функција, која узима ,,нормалне" аргументе и враћа ,,нормални'' резултат, постави у контекст апликативног функтора и примени на аргументе, који су унутар истог тог контекста: `pure(f) <*> a1 <*> a2 <*> a3...`

Израз `pure(f)<*> a` може се написати као `f 'fmap' a`. У модулу `Applicative` дефинисан је помоћни оператор `<$>`, који је синоним за `fmap`. Имајући ово у виду, претходни кôд се може написати мало другачије, у свом идиоматском облику: `f <$> a1 <*> a2 <*> a3...`

Ова техника се назива апликативни стил, јер је аналогна обичној примени (апликацији) функција. Свака монада је апликативни функтор, иако класа `Monad` тренутно не наслеђује класу `Applicative`, из споменутих историјских разлога. Оператор `<*>` се може имплементирати коришћењем оператора `>>=`. Обрнуто не

важи. Могуће је написати инстанце апликативног функтора које не могу бити инстанце монаде. Ми се у овом раду нећемо бавити таквим апликативним функторима, већ ћемо за сваку имплементирану монаду написати оператор `<*>`, како би дати тип могао да се користи у оквиру апликативног стила, када је то погодније.

4.3 Имплементација у програмском језику F#

Језик F# нема могућност дефинисања класа типова јер полиморфизам вишег рода, који је за то неопходан, у овом тренутку не би био компатибилан са остатком .NET платформе. Уместо тога, F# користи израчунљиве изразе (Computation Expressions) [25]. Да би се неки тип могао користити као монада, неопходно је дефинисати класу која ће за тај тип имплементирати `Bind` и `Return` функције, које су аналогне Haskell `return` и `>>=` функцијама. Након тога је монаду могуће користити унутар израчунљивог израза, користећи `let!` команду за везивање монадског резултата за променљиву, односно `return` команду за позивање `Return` функције, што је аналогно Haskell `do` нотацији. Коришћење израчунљивих израза за имплементацију апликативног функтора није подржано у званичној верзији компајлера, али је могуће кроз истраживачку екстензију описану у [25].

4.4 Монаде у језику Swift

4.4.1 Детаљи имплементације

Сваку од монада `M<T>` имплементирамо на следећи начин:

1) Дефинишемо методтан у стандардној библиотеци за дату монаду;`map<U>(f:T->U)->M<U>` уколико тај метод није већ прису-

2) Дефинишемо глобалну функцију `flatten<T>(m:M<M<T>>)->M<T>`. Функција `flatten` еквивалентна је природној операцији μ;

3) Дефинишемо глобалну функцију `pure<T>(value:T),` `—>M<T>` која је еквивалентна природној операцији η и Haskell функцији `return`

4) Дефинишемо метод `bind<U>(f:T->M<U>)->M<U>`, који је аналоган Haskell, аналоган На-`>>=` оператору и F# `Bind` функцији и метод `bind<U>(m:M<U>)->M<U>` skell `>>` оператору.

Функција `pure` (3) није увек неопходна у пракси, јер се уместо ње могу користити конструктори или синтаксичке пречице уколико постоје за дати тип, али је ипак наводимо ради комплетности. Уколико није другачије назначено, `bind` методе (4) имплементирамо користећи претходно дефинисане `map` и `flatten` операције:

```
extension M { func bind<U>(f: T -> M<U>)
   -> M<U> { return flatten(self.map(f))
```

```
    }

    func bind<U>(monad: M<U>) -> M<U> { return
        self.bind { _ in monad }
    }
}
```

Ова имплементација је елегантна јер следи из природних монадских операција, али није увек најефикаснија у Swift језику. Због тога смо за монаду `Array<T>` написали ефикаснију императивну имплементацију. Најновија верзија језика, Swift 1.2, која је још увек у бета фази развоја, укључује у стандардну библиотеку метод `flatMap`, на типовима `Array<T>` и `Optional<T>`. Овај метод је семантички исти као метод `bind` представљен у овом раду. Иако још увек нема посебне синтаксе за монаде, укључивање `flatMap` метода је знак да би то можда могло да се промени у наредним верзијама језика.

Коначно, ради коришћења датих монада у оквиру апликативног стила, имплементирали смо оператор `<^>`, који је заправо глобална `map` функција, аналогна Haskell оператору `<$>`, и оператор `<*>` аналоган истоименом Haskell оператору:

```
infix operator <^> { associativity left } infix
operator <*> { associativity left }

func <^> <T, U>(f: T -> U, monad: M<T>) -> M<U> {
    return monad.map(f)
}

func <*> <T, U>(mF: M<T -> U>, monad: M<T>) -> M<U> {
    return mF.bind { f in monad.bind { value in f(value)
        }
    }
}
```

За оператор `<*>` је, као и за метод `bind`, ова природна имплементација често недовољно ефикасна, те смо и за овај оператор написали ефикаснију императивну имплементацију за `Array<T>` монаду, док остале монаде подразумевано користе претходну дефиницују.

Функције више променљивих писали смо у облику који дозвољава парцијалну апликацију, тамо где је то неопходно (`f(a: A)(b: B)` уместо `f(a: A, b: B)`). У својој првој итерацији Swift не поседује могућност имплементације полиморфних монада као Haskell, нити поседује конструкције налик израчунљивим изразима у F#. Уколико дизајнери језика одлуче да једно од та два или нешто треће укључе у будуће верзије, биће тривијално да се имплементације, представљене у овом раду, уклопе у нову синтаксу.

4.4.2 Optional<T>

Тип `Optional<T>`, аналоган Haskell `Maybe a` типу, моделује могуће одсуство дате вредности. Део је стандардне Swift библиотеке у којој је дефинисан као унија дискриминатора, на следећи начин:

```
enum Optional<T> {
    Some(T)
    None
}
```

Ово је један од најважнијих типова у свакодневном програмирању, нарочито приликом коришћења Cocoa и Cocoa Touch оквира. Наиме, Swift нема `null` односно `nil` референце. Свака референца мора показивати на конкретни постојећи објекат у меморији. Али, с обзиром на то да су Cocoa и Cocoa Touch писани у Objective-C језику, `nil` је веома често аргумент метода или враћена вредност. Све те `nil` вредности су премошћене у `Optional<T>`. Компајлер аутоматски пакује вредности које нису `nil` у `.Some(T)`, а вредности које су `nil` у `.None`. Иако Swift нема `nil` референце, кључна реч `nil` постоји у језику и означава литерал којим се може представити било који тип који имплементира `NilLiteralConvertible` Табела 1: Синтаксичке пречице за рад са `Optional<T>` типом

основна синтакса	пречица	значење
let o: Optional<T>	let o: T?	декларација
let o: T? = .Some(5)	let o: T? = 5	додела
let o: T? = .None	let o: T? = nil	додела
switch o {case .Some(let v):}	if let v = o {}	отпакивање
if let v = o {v.method()}	v?.method()	уланчавање

протокол, а `Optional<T>` је један од њих. У пракси је најчешће `nil` синоним за `Optional.None`. Уграђене синтаксичке пречице за рад са `Optional<T>` наведене су у Табели 1.

`Optional<T>` чини рад са референцама далеко безбеднијим и отклања читаву класу могућих грешака у коду. Поред тога, тип је генералнији од `null` референци јер може изразити и одсуство типова који нису референце, већ вредности. Ипак, и поред споменутих синтаксичких пречица, понекад рад за овим типом може довести до незграпног кода са угњежденим `if` блоковима, који је тежак за читање и одржавање. Монадска апстракција може бити добра алтернатива у таквим случајевима. Метод `map` је већ дефинисан у стандардној библиотеци за овај тип. Операција мапирања враћа `.None` ако је `Optional<T>` једнак `.None`. У супротном, примењује дату функцију $f: T \rightarrow U$ је тривијална - узима дату вредност и враћа је на садржај `Optional<T>` и враћа резултат као `Optional<U>`. Функција `pure` запаковану у `Optional<T>`. Функција `flatten` враћа унутрашњи `Optional<T>` или `.None`, уколико је спољашњи `Optional<T>` једнак `.None`:

```
func pure<T>(value: T) -> T? { return
   value
}
```

```
func flatten<T>(optional: T??) -> T? { if
    let innerOptional = optional { return
    innerOptional
    } else {
       return nil
    }
}
```

Монадски `bind` методи, апликативни оператор `<*>` и оператор `<^>` користе претходно описану подразумевану имплементацију.

Уграђени `?.` оператор за уланчавање (Табела 1) је заправо синтаксичка пречица за `bind` метод:

```
let maybeC1 = a?.b()?.c()
let maybeC2 = a.bind { $0.b() }.bind { $0.c() }
```

Предност `bind` је што се може користити са произвољним функцијама и методима, укључујући и ламбда изразе за једнократну употребу. Примена овог метода у интерактивном Swift окружењу приказана је на Сл. 16.а.

Без `bind` метода, решавање сличног проблема, у тренутној верзији језика, захтева коришћење угњеждених `if-let` блокова. Додавање сваке следеће функ-

```
typealias BinaryOperation = (Double, Double) -> Double

let availableOperations: [String : BinaryOperation] = [
    "+" : (+),
    "-" : (-),
    "*" : (*),
    "/" : (/)
]

func perform
    (operation: String)
    (_ argument1: Double) (_ argument2: Double) -> Double? {
        return availableOperations[operation]?(argument1, argument2)
}

let result =
perform ("+") (2)(3)
    .bind { result in perform ("*") (result)(3) }
    .bind { result in perform ("/") (result)(2) }

println(result)
```

```
["-": (Function), "+": (Function), "*": (Function), "/": (Function)]
(5 times)
7.5
1.5e+1
7.5
"Optional(7.5)"
```

(a)

```
func max(a: Double)(b: Double) -> Double {
    return a > b ? a : b
}

let greater = max <^> perform ("/") (5)(2) <*> perform ("-") (10)(7)
println(greater)
```

```
3.0
3.0
"Optional(3.0)"
```

(b)

Слика 16. (а) Примена монадског `bind` метода за уланчавање операција које могу вратити `nil` уколико бинарна операција не постоји за дати симбол; (b) Примена апликативног `<*>` оператора за рад са независним операцијама, од којих свака може вратити `nil`

ције у ланац `if-let` везивања, захтева додатно угњеждење, што врло брзо по-методачитљивостостајепотпунонечитљиво,докприликомкоришћења`bind` је иста за било који број додатих функција.

Апликативни стил је погоднији када опциони резултати не зависе један од другог, већ су независни аргументи неке функције (Сл. 16.b). Слично `bind` методу, предност апликативног стила нарочито долази до изражаја када функција, која се примењује на опционе вредности, има много аргумената.

4.4.3 Result<T>

Због аутоматског бројања референци, ухватљиви изузеци нису погодни за рад са операцијама које могу резултирати грешком у Swift и Objective-C језицима. Уместо тога, Cocoa и Cocoa Touch оквири дефинишу `NSError` класу која енкапсулира информације о грешци [26]. Пошто методи у Objective-C, језику у којем су написани Cocoa и Cocoa Touch, немају могућност да врате више од једне вредности, `NSError` се користи тако што се прослеђује по референци:

```
var maybeError: NSError?
let failableResult = readFile(&maybeError)
if let result = failableResult { // use
result
} else if let error = maybeError {
   // deal with error
}
```

Проблем претходног кода је што је поприлично неелегантан у Swift језику, нарочито ако је неопходно проћи кроз ланац операција од којих свака може да резултује грешком. Једно могуће решење је да се као резултат операције врати пар, који садржи резултат или грешку. `func readFile() -> (Result?, NSError?)`

Проблем овог приступа је што пар и генерално n-торка не представља добро резултат који може бити или успех или грешка, али не и оба. Поред тога, резултат и грешка морају нужно да буду упаковани у `Optional<T>`. Решења оба проблема је алгебарски тип `Result<T>`:

```
class Box<T> { let
   value: T init(_
   value: T) {
   self.value = value
   }
}

enum Result<T> { case
   Success(Box<T>) case
   Error(NSError) init(_ value:
   T) { self =
   .Success(Box(value))
   }
}
```

Класа `Box<T>` је неопходна јер Swift компајлер у верзији 1.1 не дозвољава енумерације које, поред генеричке придружене вредности (у овом случају `T`), садрже и негенеричке (у овом случају `NSError`). Када се `Result<T>` користи у монадском или апликативном стилу, ово не представља велики проблем, јер тада одговарајући

оператори и методи обављају отпакивање и запакивање иза сцене. Операцију `map` над типом `Result<T>` дефинисали смо на следећи начин:

```
func map<U>(f: T -> U) -> Result<U> {
   switch self { case .Success(let
   boxedValue):
      return .Success(Box(f(boxedValue.value)))
   case .Error(let error):
      return .Error(error)
   }
}
```

Уколико је `Result<T>` једнако `Success`, тада `map` узима придружени резултат, примењује прослеђену функцију на тај резултат и враћа нови резултат, запаковану`Result<T>`.Монадскаоперација`pure` јетривијална-узимапрослеђенувредност `a` и враћа `.Success(a)`. Функција `flatten` враћа унутрашњи `Result<T>` или `.Error(e)`, уколико је спољашњи `Result<T>` једнак `.Error(e)`:

```
func pure<T>(value: T) -> Result<T> { return
   .Success(Box(value))
}

func flatten<T>(r: Result<Result<T>>) -> Result<T> {
   switch r {
   case .Success(let boxedResult):
      return boxedResult.value
   case .Error(let error):
      return .Error(error)
   }
}
```

Као и у случају `Optional<T>`, монадски `bind` методи, апликативни оператор `<*>` и оператор `<^>`, користе подразумевану имплементацију.

Нови методи и функције се могу од самог почетка дизајнирати тако да враћају `Result<T>`, а за већ постојеће се могу написати помоћне функције на следећи начин:

```
func readFile() -> Result<String> { var
   maybeError: NSError?
   let failableResult = readFile(&maybeError)
   if let result = failableResult { return
   Result(result)
   } else if let error = maybeError { return
      .Error(error)
   }
}
```

Функција `perform`, коју смо користили да илуструјемо третирање `Optional<T>` као монаде (Сл. 16.a), може се имплементирати тако да, уместо опционог резултата, враћа `Result<T>` (Сл. 17.a).

```
205 func perform
206     (operation: String)
207     (_ argument1: Double) (_ argument2: Double) -> Result<Double> {
208         if let operation = availableOperations[operation] {
209             return Result(operation(argument1, argument2))
210         }
211         return .Error(OperationNotFound)
212 }
213
214 let result: Result<Double> =
215 perform ("+") (2)(3)
216     .bind { result in perform ("*") (result)(3) }
217     .bind { result in perform ("/") (result)(2) }
218
219 println(result)
```

(5 times)

(Enum Value)

(Enum Value)
(Enum Value)

"Result(7.5)"

(a)

```
221 let greater: Result<Double> = max <^> perform ("/") (5)(2) <*> perform ("-") (10)(7)
222 println(greater)
```

(Enum Value)
"Result(3.0)"

(b)

Слика 17. (a) Примена монадских `bind` метода за уланчавање операција које могу вратити `Result.Error` уколико бинарна операција не постоји за дати симбол; (b) Примена апликативног `<*>` оператора за рад са независним операцијама, од којих свака може вратити `Result.Error`

Кôд приказан на Сл. 17. је практично идентичан еквивалентном коду за тип `Optional<T>`. Овде се види природност монадске апстракције у функционалним језицима, чак и ако ти језици не поседују специјалну синтаксу за монаде. Исто важи и за апликативни стил. Функција `max`, дефинисана у коду приказаном на Сл. 16.b, може се користити са `Result<T>` на потпуно исти начин као са `Optional<T>` (Сл. 17.b).

4.4.4 Array<T>

Третирањем низова као монада могу се имплементирати операције чији је резултат недерминистички, тј. оне операције код којих се, уместо једног тачно одређеног резултата, добија више могућих. Операција `map` је већ дефинисана за `Array<T>` у стандардној библиотеци као метод, те оператор `<^>` једноставно позива тај метод. Монадска операција `pure` узима дати елемент и враћа га упакованог у низ, док операција `flatten` низ низова трансформише у један низ, користећи метод `reduce`:

```
pure<T>(element: T) -> [T] { return

    [element]

}


func flatten<T>(array: [[T]]) -> [T] { return

    array.reduce([]) { $0 + $1 }

}
```

Имплементације монадских метода и апликативног оператора би могле једноставно да се изведу из претходних функција, али та имплементација не би била довољно

ефикасна јер би више пута пролазила кроз исти низ. Због тога смо за `bind` и `<*>` написали ефикасније императивне имплементације:

```
extension Array { func bind<U>(f: T ->
    [U]) -> [U] { var result = [U]()
    for element in self { result +=
    f(element)
        } return
        result
    }

    func bind<U>(array: [U]) -> [U] { return

        self.count > 0 ? array : []

    }
}

public func <*><T, U>(fs: [T -> U], array: [T]) -> [U] {
    var results = [U]() for f in fs { for element in array
    { results += [f(element)]
        } } return
    results
}
```

На Сл. 18. приказана је примена монадских метода и апликативног оператора на низове, у интерактивном Swift окружењу.

```
36 func toCharacters(string: String) -> [Character] {
37     return Array(string)
38 }
39
40 let languages = ["Swift", "Haskell", "Java"]
41 let characters = languages.bind(toCharacters)
```

```
(3 times)

["Swift", "Haskell", "Java"]
["S", "w", "i", "f", "t", "H", "a", "s", "k", "e", "l", "l", "J", "a", "v", "a"]
```

(a)

```
43 let ranks = [
44     "A", "2", "3", "4", "5", "6", "7",
45     "8", "9", "10", "J", "Q", "K"
46 ]
47 let suits = ["❤️", "♦️", "♠️", "♣️"]
48
49 func card(rank: String)(suit: String) -> String {
50     return rank + suit
51 }
52 let deck = card <^> ranks <*> suits
```

```
["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]

["❤️", "♦️", "♠️", "♣️"]

(52 times)

["A❤️", "A♦️", "A♠️", "A♣️", "2❤️", "2♦️", "2♠️", "2♣️", "3❤️",...
```

(b)

Слика 18. (a) Примена монадског `bind` метода за растављање низа ниски у низ карактера; (b) Примена апликативног `<*>` оператора за конструкцију свежња карата из низа знакова и вредности

5 Дизајн фунционалне библиотеке за компрехенсију листа

Компрехенсија листа (List Comprehension) је синтаксичка конструкција за једноставно извођене нових листа (или сродних типова података попут низова, секвенци итд.) из већ

постојећих. У Haskell језику ова синтакса изгледа слично математичкој нотацији за скупове [27]:

```
divisibleByThree = [ x | x <- [1..1000], x `mod` 3 == 0]
```

0}Претходни кôд је еквивалентан математичкој нотацији. Еквивалентна синтакса у F#

језику [25] слична је петљама у императивним$\{x|x \in [1,100] \wedge xmod3$ = језицима:

```
let divisibleByThree = [ for x in 1 .. 1000 do if x % 3 = 0 then yield
   x ]
```

Сличност са императивним кодом је само синтаксичка. Претходна `for` конструкција није наредба, већ израз, чија је вредност нова листа. Сличне синтаксичке конструкције поседују и други језици, као што су Python, C# итд. Swift тренутно не поседује специјализовану синтаксу за ту намену. Када се ради само са једном листом, компрехенсију је могуће, а често и боље, заменити функцијама вишег реда, као што су `map` и `filter`. Компрехенсија више од једне листе може се заменити угњежденим монадским везивањем. Заправо, компрехенсија листа је само синтаксичка пречица за монадско везивање. За ислустрацију те чињенице користимо наивни алгоритам за проналажење свих Питагориних тројки из бројева од 1 до 100:

```
[ (x, y, z) | z <- [1..100], y <- [1..z], x <- [1..y], x * x + y * y ==

   z *z ]
```

Претходна компрехенсија је еквивалентна следећем монадском везивању:

```
[1..100] >>= \z ->
   [1..z] >>= \y —>
      [1..y] >>= \x —>
         guard(x * x + y * y == z * z) >> return (x, y, z)
```

Коришћењем `do` конструкције, претходно везивање може се написати на читљивији начин:

```
do let z = [1..100] let y =
   [1..z] let x = [1..y] guard(x
   * x + y * y == z * z) return
   (x, y, z)
```

С обзиром на то да Swift не поседује посебну синтаксу за монаде, `bind` метод је једини начин имплементације претходног Haskell кода:

```
Array(1...100).bind { (z: Int) in
   Array(1...z).bind { (y: Int) in
      Array(1...y).bind { (x: Int) in
         x * x + y * y == z * z ? [(x, y, z)] : []
      }
   }
}
```

У делу 4.4.4 смо показали да се, коришћењем `bind` метода, може знатно упростити уланчавање функција које враћају низове. Међутим, корист од овог метода је упитна у случајевима као што је проналажење Питагориних тројки. У том и сличним случајевима, једина предност у односу на императивни кôд је то што је `bind` чиста функција, те се о коду може лакше резоновати, исправност кода се може лакше доказати и кôд се може знатно лакше паралелизовати. Међутим, читљивост кода није боља, а у неким случајевима је и гора, од читљивости кода који користи `for` петље. Поред тога, угњеждене `bind` методе доводе до знатног пада перформанси, када се ради са великим низовима. За ова два проблема не постоји потпуно задовољавајуће решење у општем случају. Идеално решење би била синтаксичка конструкција за секвенце, аналогна постојећој конструкцији за опционо везивање, и слична изразима секвенци из F# језика. Коришћењем такве хипотетичке конструкције, претходни кôд би се могао написати знатно елегантније:

```
for z in 1...100, y in 1...z, x in 1...y where x * x + y * y == z * z
    { yield (x, y,
    z)
}
```

Овакву конструкцију није могуће написати унутар тренутног језика, већ искључиво као екстензију компајлера. Пошто је кôд компајлера тренутно затворен, морамо се задовољити компромисним решењима. Као што смо рекли, опште решење није могуће написати. Угњеждене `bind` методе су неопходне у случају када секвенце са којима се ради зависе једна од друге. На срећу, то је у пракси далеко ређе од рада са независним секвенцама. У том случају, када су секвенце независне једна од друге, могуће је написати функције чије је коришћење готово подједнако елегантно као претходно предложена хипотетичка `for` конструкција. У овом поглављу описујемо дизајн и коришћење библиотеке таквих функција.

Поред тога што представља решење конкретног проблема компрехенсије листа, ова библиотека ислуструје неке опште проблеме, који се могу појавити при писању других функционалних библиотека у језику Swift, и могућа решења тих проблема.

5.1 Алтернативна имплементација и генерализација апликативног стила

У случају рада са међусобно независним листама и у одсуству услова филтрирања, угњеждене `bind` методе је могуће заменити апликативним стилом. Уколико, уместо јединствених Питагориних тројки, желимо да формирамо све могуће уређене тројке из бројева од 1 до 100, монадски код за то се може написати на следећи начин:

```
Array(1...100).bind { (z: Int) in
    Array(1...100).bind { (y: Int) in
        Array(1...100).bind { (x: Int) in
            [(x, y, z)]
        }
    }
}
```

Написан апликативним стилом, претходни кôд се може свести на једну линију и помоћну функцију:

```
func triple<X, Y, Z>(x: X)(y: Y)(z: Z) -> (X, Y, Z) {
   return (x, y, z)
} triple <^> Array(1...100) <*> Array(1...100) <*>
Array(1...100)
```

Ово је знатно краће и читљивије од монадског кода, али има неколико недостатака:

1. Сваки појединачан позив апликативног оператора доводи до копирања целокупног претходног резултата, што даље доводи до лоших перформанси за велике низове;
2. Апликативни стил не омогућава филтрирање резултата у току саме апликативне операције, већ се крајњи резултат мора експлицитно пропустити кроз `filter` функцију. То доводи до још једног копирања и додатног пада перформанси;
3. Апликативни стил је директно преузет из Haskell-а, где је опште познат. С другестране, програмерима који користе Swift, значење уведених оператора може бити потпуно непознато, што може да буде психолошка препрека ка прихватању овог стила програмирања;
4. Функција, која се прослеђује апликативном оператору, мора бити написана уоблику који омогућава парцијалну апликацију. Иако Swift природно подржава парцијалну апликацију, она није подразумевана, већ функције морају да буду написане на посебан начин, да би их било могиуће парцијално позвати. Ово онемогућава директну примену већ постојећих функција, без претходне обраде.

Прва два проблема би се могла делимично решити имплементирањем апликативног оператора на лењим секвенцама, уместо на низовима. Такво решење би било задовољавајуће за многе примене, али би перформансе и даље биле слабије од оптималних. Циљ библиотеке коју имплементирамо су перформансе практично идентичне императивном коду, уз све погодности функционалне парадигме. Решење сва четири проблема се може добити генерализацијом алтернативног облика апликативног стила.

5.1.1 Имплементација еквивалената `liftA[n]` односно `liftM[n]` функција за секвенце

Функције `liftA[n]` односно `liftM[n]` се користе за алтернативни облик апликативног стила у Haskell-у. Последњих n аргумената ових функција су n апликативних функтора (`liftA[n]`) односно монада (`liftM[n]`), а први аргумент је функција n аргумената, чији типови морају да се подударају са унутрашњим типовима апликативних функтора, односно монада, и то истим редом. Суштински је неопходна само једна од ове две фамилије функција, али, пошто тренутно класа монада не наслеђује класу апликативни функтор, обе функције су присутне у стандардним библиотекама и то `liftA[n]` до n = 3, `liftM[n]` до n = 5. Захваљујући полиморфизму више врсте, ове функције могу да се користе са било којим апликативним функторима, односно монадама, а не само са листама. Ми се, за потребе овог рада, ограничавамо

на примену датих функција на листе. Користећи `liftA3`, односно `liftM3` функцију, формирање свих уређених тројки из бројева од 1 до 10 може се написати на следећи начин:

```
liftA3 (,,) [1..10] [1..10] [1..10] liftM3
(,,) [1..10] [1..10] [1..10]
```

Испоставља се да је ова фамилија функција најоптималнији модел за имплементацију апликативног стила у Swift-у. Имплементација коју излажемо у овом раду има следеће карактеристике:

1. Име сваке функције је `foreach`, уместо `liftX[n]`. Сматрамо да ово име боље описује намену функција за компрехенсију листа. Такође, захваљујући томе што је у Swift-у дозвољено исто име за функције са различитим бројем и типом аргумената и враћених вредности, није неопходан суфикс који указује на то са колико листа функција ради. Ово омогућава једноставнији и јединствени јавни интерфејс.

2. Функције нису ограничене на низове, већ могу да се користе са било којим секвенцама, то јест са било којим типовима који имплементирају `SequenceType` протокол. Функције нису лење и одмах враћају цео крајњи резултат, као низ. Могуће је написати лење верзије, које враћају секвенцу уместо низа. Те верзије би, међутим, имале слабије перформансе у случајевима када јесте неопходан цео резултат, јер тренутна верзија компајлера доста боље оптимизује низове него секвенце. Функције су генеричке, те се истовремено може се радити са секвенцама различитих типова, што је веома корисно у пракси.
3. Приватна имплементација сваке функције је императивна и користи променљиви акумулаторски низ за чување међурезултата. Ово је кључно за одржавање добрих перформанси. Јавни интерфејс ових функција је чисто функционалан.

4. Функцију којатрансормише појединачне елементе секвенцинаводимо као последњи аргумент `foreach` функција. Ово омогућава да се, приликом позивања, користи погодна синтакса (Trailing Closure Syntax).

5. У овом раду смо имплементирали функције које раде са до три секвенце, али јетривијално генерализовати их, тако да раде са произвољним бројем секвенци.

Имплементација `foreach` функција приказана је следећим кодом:

```
func foreach<
   S: SequenceType,
   T>( s: S,
   #yield: (
   S.Generator.Element
      —> T)
   )—> [T] {
      var result = [T]()
      for e in s {
         result.append(yield(e))
      } return
      result
}
```

```
func foreach<
    S1: SequenceType,
    S2: SequenceType,
    T>( s1: S1, s2:
    S2, #yield: (
    S1.Generator.Element,
    S2.Generator.Element) -> T)

    _
      > [T] { var result =
        [T]() for e1 in s1
        {
            for e2 in s2 {
                result.append(yield(e1, e2))
            } } return
        result
}

func foreach<
    S1: SequenceType,
    S2: SequenceType,
    S3: SequenceType,
    T>( s1: S1, s2:
    S2, s3: S3,
    #yield: (
    S1.Generator.Element,
    S2.Generator.Element,
    S3.Generator.Element) -> T)

    -> [T] { var result =
        [T]() for e1 in s1
        {
            for e2 in s2 {
                for e3 in s3 {
                    result.append(yield(e1, e2, e3))
                }
            } } return
        result
}
```

Прва `foreach` функција, која узима једну секвенцу као аргумент, идентична је глобалној `map` функцији, те није неопходна у пракси. Наводимо је ради комплетности, пошто су остале `foreach` функције њена генерализација.

```
let triples = foreach(1...10, 1...10, 1...10) { ($0, $1, $2) }
println(triples)
```

```
(1001 times)
"[(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 1, 4), (1, 1, 5), (1, 1, 6), (1, 1,...
```

(a)

```
let ranks = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
let suits = "♥♦♠♣"

let cards = foreach(ranks, suits) { $0 + String($1) }
println(cards)
```

```
["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
"♥♦♠♣"

(53 times)
"[A♥, A♦, A♠, A♣, 2♥, 2♦, 2♠, 2♣, 3♥, 3♦, 3♠, 3...
```

(b)

Слика 19. (a) Примена `foreach` функције за формирање свих уређених тројки из бројева од 1 до 10; (b) Примена `foreach` функције за конструисање шпила карата из ниске знакова и низа вредности.

На Слици 19.a показано је коришћење претходно дефинисаних функција за конструињање уређених тројки из бројева од 1 до 10. На Слици 19.b приказано је конструисање шпила карата из знакова и ниски. Овај пример је аналоган примеру апликативног стила са слике 18.b и крајњи резултат је исти. Једина разлика је то што су у примеру на Слици 19.b знакови представљени једном ниском, уместо низом ниски, како бисмо показали да се `foreach` функције могу користити са било којим секвенцама, а ниске у Swift-у су секвенце карактера.

Овим су решена три од четири проблема апликативног стила, описана у делу 5.1. Перформансе `foreach` функција су практично идентичне императивном коду. Функције су разумљиве било ком Swift програмеру, јер не представљају само директну копију имплементације из Haskell језика, већ су прилагођене синтакси Swift-а. Коначно, `foreach` функције се могу користити са било којим функцијама трансформације, укључујући и оне већ постојеће, уколико је њихов тип одговарајући. Мана овог приступа је понављање кода, јер је сваку `foreach` функцију неопходно посебно имплементирати. Додатна мана, у односу на коришћење апликативног оператора, је то што се у овом приступу не може радити са произвољним бројем функција. Обе мане углавном нису значајне у свакодневном раду. Функције је неопходно имплементирати само једном и додавање нових је тривијално. Такође, у пракси је најчешће неопходно радити са свега неколико листа у једном тренутку, те је десетак `foreach` функција више него довољно. У наредним деловима предлажемо решење четвртог проблема, уводећи могућност филтрирања резултата.

5.1.2 Филтрирање резултата компрехенсије коришћењем `Yield<T>` типа

Филтрирање резултата се може постићи у `foreach` функцијама ако трансформационе функције не врате увек вредност која се додаје у резултујући низ, већ само у случају испуњеног услова. Постоји неколико начина да се то постигне.

Једна могућност је да трансформационе функције врате `Optional<T>` тип, који моделује могуће одсуство вредности. Ипак, `Optional<T>` није оптимално решење у овом случачају. Корисник библиотеке би морао експлицитно да врати `Optional.None`, сваки пут када жели да искључи одређену вредност из крајњег резултата. Оптималније решење, које је више у духу декларативне парадигме, је да се експлицитно наведе услов при којем се дата вредност укључује у крајњи резултат, а да у супротном вредност буде имлицитно искључена. Ово се може постићи коришћењем описнијег `Yield<T>` типа, који садржи и услов и вредност. Овим типом се јасније исказује семантика функције трансформације, у односу на коришћење генеричког `Optional<T>` типа. У функционалном програмирању је генерално добра пракса користити посебно описне типове за моделовање домена у којем се ради. Тиме типови служе као документација и смањује се потреба за експлицитним коментарима у коду.

Следећа одлука, која се мора донети, је дизајн самог `Yield<T>` типа. Овај тип би у принципу могао бити обичан пар типа `(Bool, T)`, али је је бољи избор неки од потпунијих типова, дакле енумерација, структура или класа. Класе, као типови референци, имају нешто слабије перформансе од енумерација и структура. Коришћење

класе није неопходно у овом случају јер није потребно да инстанце `Yield<T>` типа буду референце, већ вредности. Избор између енумерација (унија дискриминатора) и структура није очигледан, те ћемо подробније анализирати обе опције. Енумерација `Yield<T>` се може имплементирати на следећи начин:

```
enum Yield<T> { case Value(() -> T) case
   ValueIfCondition(() -> Bool, () -> T)

   init(_ value: () -> T) { self
      = .Value(value)
   }

   func ifCondition(condition: () -> Bool) -> Yield<T> { switch
      self {
         case .Value(let value):
            return .ValueIfCondition(condition, value)
         case let .ValueIfCondition(_, value):
            return .ValueIfCondition(condition, value)
      }
   }
}
```

Вредност која се укључује у крајњи резултат је враћена вредност функције типа `()->T`, уместо да буде сам тип `T`, како би израчунавање вредности било одложено до провере услова. Ово није неопходно у Haskell-у, у којем је заступљено лењо израчунавање, али јесте нопходно у практично свим осталим језицима, па тако и у Swift-у и F#. Услов је такође представљен функцијом типа `()->Bool`. Ово није строго неопходно, за разлику од вредности, али је корисно зарад побољшања перформанси. Ако се `Yield<T>` имплементира као структура, услов и вредност постају својства инстанци:

```
struct Yield<T> { let
   condition: () -> Bool let
   value: () -> T

   init(_ value: () -> T) { self.init({
      true }, value)
   }

   private init(_ condition: () -> Bool, _ value: () -> T) {
      self.condition = condition self.value = value
   }
```

```
    func ifCondition(condition: () -> Bool) -> Yield<T> { return

       YieldS(condition, value)

    }
}
```

Да бисмо вредност, садржану у типу `Yield<T>`, додали у крајњи резултат, написали смо помоћни метод на типу `Array<T>`, за оба случаја:

1. За додавање вредности `Yield<T>` структуре:

```
extension Array { mutating func append(yield: Yield<T>) {
   if yield.condition() { self.append(yield.value()) } }
}
```

2. За додавање вредности `Yield<T>` енумерације:

```
extension Array { mutating func
   append(yield: Yield<T>) { switch yield
   { case .Value(let value):
   self.append(value())
      case let .ValueIfCondition(condition, value):
         if condition() { self.append(value()) } }
   }
}
```

Коначно, мењамо `foreach` функцију, тако да њене `yield` функције враћају управо дефинисани `Yield<T>` тип. Захваљујући пажљиво одабраним типовима и екстензијама, успели смо да постигнемо да једина разлика, у односу на првобитну имплементацију `foreach` функција, буде то да функција трансформације враћа `Yield<T>`, уместо `T`:

```
func foreach<
   S: SequenceType,
   T>( s: S,
   #yield: (
   S.Generator.Element
   ) -> Yield<T>)
   _
    > [T] { var result =
      [T]() for e in s {
         result.append(yield(e))
      } return
      result
}

func foreach<
   S1: SequenceType,
   S2: SequenceType,
   T>( s1: S1, s2:
   S2, #yield: (
```

```
    S1.Generator.Element,
    S2.Generator.Element) -> Yield<T>)

    _
      > [T] { var result =
        [T]() for e1 in s1
        {
           for e2 in s2 {
              result.append(yield(e1, e2))
           } } return
        result
}

func foreach<
    S1: SequenceType,
```

Табела 2: Перформансе императивног кода при формирању свих уређених тројки (T1), Питагориних тројки (T2) и јединствених Питагориних тројки (T3) из разних опсега бројева. Време извршавања кода изражено је у секундама.

опсег	T1	T2	T3
1…10	0.00010	0.00006	0.00006
1…100	0.03022	0.00118	0.00033
1…1000	88.19540	0.72901	0.12887

```
    S2: SequenceType,
    S3: SequenceType,
    T>( s1: S1, s2:
    S2, s3: S3,
    #yield: (
    S1.Generator.Element,
    S2.Generator.Element,
    S3.Generator.Element) -> Yield<T>)

    -> [T] { var result =
        [T]() for e1 in s1
        {
           for e2 in s2 {
              for e3 in s3 {
                 result.append(yield(e1, e2, e3))
              }
           } } return
        result
}
```

За корисника библиотеке, коришћење обе варијанте је идентично, те су перформансе у овом случају главни критеријум за избор између две опције. За тестирање перформанси смо користили формирање свих тројки, свих Питагориних тројки и свих јединствених Питагориних тројки из три различита опсега бројева. Резултати, наведени у табелема 2, 3 и 4, показују да су перформансе енумерација неприхватљиве за велике улазе, док су перформансе структура праткично идентичне перформансама императивног кода.

Посебно занимљив случај је формирање јединствених Питагориних тројки. У императивном решењу се дупликати уклањају у току саме итерације. Еквивалентан функционални кôд би се могао написати коришћењем `bind` метода, али не и коришћењем `foreach` функције. У другом случају се мора проћи кроз све бројеве и дупликате је неопходно експлицитно филтрирати. То нужно доводи до смањења перформанси у односу на императивни кôд, али је то смањење незнатно. Чак и за тако велики улаз од 1 000 000 000 бројева, време неопходно за решавање проблема је мање од једне секунде. Према томе, `foreach` функције у неким случајевима могу бити добра алтернатива угњежденим `bind` методима.

Коначно, ради упрошћавања јавног API, користићемо својство Swift-а које се

Табела 3: Перформансе `foreach` функције, која користи `Yield<T>` енумерацију, при формирању свих уређених тројки (T1), Питагориних тројки (T2) и јединствених Питагориних тројки (T3) из разних опсега бројева. Време извршавања кода изражено је у секундама.

опсег	T1	T2	T3
1…10	0.00012	0.00053	0.00051
1…100	0.02714	0.33006	0.30933
1…1000	86.25757	322.39997	318.25416

Табела 4: Перформансе `foreach` функције, која користи `Yield<T>` структуру, при формирању свих уређених тројки (T1), Питагориних тројки (T2) и јединствених Питагориних тројки (T3) из разних опсега бројева. Време извршавања кода изражено је у секундама.

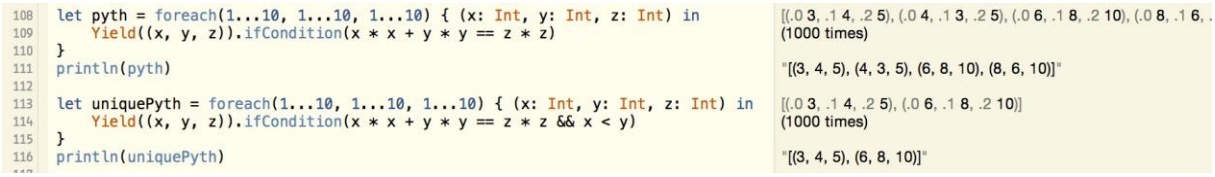
опсег	T1	T2	T3
1…10	0.00009	0.00005	0.00009
1…100	0.02898	0.00105	0.00105
1…1000	78.54783	0.61933	0.61665

назива `autoclosure`. Тиме се постиже да корисник библиотеке може да користи обичне вредности, које компајлер иза сцене аутоматски пакује у функције, како би одложио њихово израчунавање. Тиме добијамо коначну имплементацију `Yield<T>` структуре:

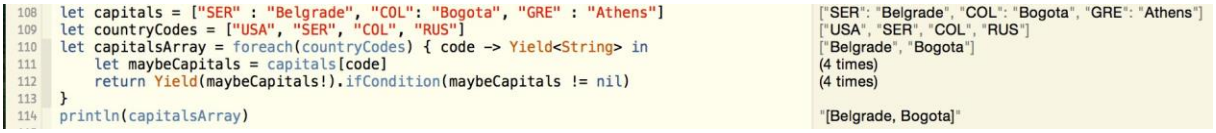
```
struct Yield<T> { let
   condition: () -> Bool let
   value: () -> T

   init(@autoclosure(escaping) _ value: () -> T) {
      self.init({true}, value)
   }

   private init(_ condition: () -> Bool, _ value: () -> T) {
      self.condition = condition self.value = value
   }
```

```
    func ifCondition(@autoclosure(escaping) condition: () -> Bool) ->
        Yield<T> { return Yield(condition, value)
    }
}
```

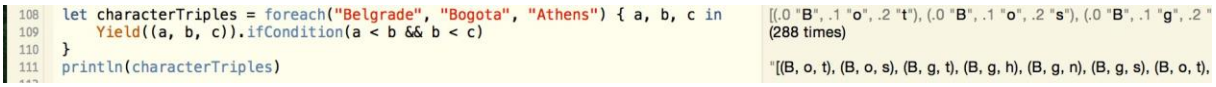
На сликама 20 - 22 приказана је примена дефинисаних функција за различите компрехенсије секвенци, укључујући проналажење Питагориних тројки, које је коришћено за тестирање.



Слика 20. Примена `foreach` функција за проналажење Питагориних тројки



Слика 21. Примена `foreach` функција за проналажење вредности у речнику, за дати низ кључева



Слика 22. Примена `foreach` функција за манипулацију индивидуалних карактера ниски

5.1.3 Враћање више од једне вредности

Имплементација `foreach` функција и `Yield<T>` типа, изложена у претходном делу, омогућава да се у крајњи резултат укључи само једна вредност за сваку појединачну комбинацију елемената секвенци. Више вредности је једино могуће вратити као низ низова. Међутим, често је пожељно да се више вредности укључи појединачно у крајњи резултат, уместо као низ. У F# језику се то може постићи командом `yield!`:

```
[for x in 1 .. 5 do yield! [x; x * x] ]
```

Резултат претходне операције је листа [1;1;2;4;3;9;4;16;5;25]. Желимо да додамо сличну функционалност у `foreach` функције, које смо имплементирали. Једна могућност је да проширимо `Yield<T>` тип, тако да чува низ, уместо појединачне вредности, али би такав приступ смањио перформансе. Боље решење је да напишемо верзију метода `append`, на типу `Array<T>`, који као аргумент узима `Yield<[T]>`, уместо `Yield<T>`:

```
extension Array { mutating func append(yield: Yield<[T]>)
    { if yield.condition() { self.extend(yield.value()) } }
}
```

У библиотеку је неопходно додати верзију `foreach` функција, које враћају вредност `Yield<[T]>`. Осим промене враћеног типа функције трансформације, није неопходно променити ништа више унутар имплементације:

```
func foreach<
```

```
    S: SequenceType,
    T>( s: S,
    #yield: (
    S.Generator.Element
    ) -> Yield<[T]>)

    -
      > [T] { var result =
        [T]() for e in s {
            result.append(yield(e))
        } return
        result
}
```

Компајлер ће, након овог додатка, сваки пут када корисник библиотеке врати `Yield<[T]>`, позвати одговарајућу верзију `append` метода и додати појединачно враћене вредности у крајњи низ. Уколико корисник заиста жели да добије низ низова, потребно је да експлицитно наведе тип крајњег резултата као `[[T]]`. Оба ова случаја илсутрована су на Слици 23.

```
108  let array = foreach(1...5) { x in Yield([x, x * x]) }
109  println(array)
110
111  let arrayOfArrays: [[Int]] = foreach(1...5) { x in Yield([x, x * x]) }
112  println(arrayOfArrays)
```

```
(6 times)
"[1, 1, 2, 4, 3, 9, 4, 16, 5, 25]"

(6 times)
"[[1, 1], [2, 4], [3, 9], [4, 16], [5, 25]]"
```

Слика 23. Примена `foreach` функције за укључивање вишеструких вредности у крајњи резултат

6 Закључак

Резултатиовог рада показујупогодност Swiftјезика заједноставну имплементацију и коришћење различитих функционалних техника и апстракција. С обзиром на то да је Swift потпуно нови језик, чије спецификације још увек нису потпуно одређене, остаје отворено питање у којој мери и како ове и сличне технике из традиционалних функционалних језика треба примењивати у продукцијском коду. Једно занимљиво питање, о којем се мора размишљати приликом имплементације функционалних техника, је избор између метода и глобалних функција. Методи су, као што смо рекли, вероватно пожељни у већини случајева, али су у трентуној верзији језика глобалне функције нешто моћније јер су једини начин за имплементацију одређених генеричких апстракција. Дефнитивни одговори на ова питања ће морати да сачекају Swift 2.0, можда и 3.0 верзију. Одговори ће, између осталог, зависити и од евентуалних помоћних синтаксичких конструкција које ће језик садржати. У међувремену су, како монаде, тако и друге функционалне апстракције, несумњиво одличан начин да се упрости приватна имплементација. Међутим, те апстракције ипак треба опрезно и постепено откривати као јавни интерфејс. Јавни функционални API може да буде проблематичан, не само са техничке стране, већ и због тога што су традиционални развојни тимови углавном састављени од програмера образованих у оквиру објектно-орјентисане парадигме. Да би функционалне технике доживеле свој пуни потенцијал за повећање продуктивности, најпре је неопходна едукација и прилагођавање тимова. Различити могући начини те едукације су такође занимљив и важан практични и истраживачки проблем за данашње софтверске фирме, с обзиром на то да функционално програмирање није више само академска творевина, већ ефикасно средство највећих комерцијалних развојних плаформи.

Библиографија

[1] A. Church, ``A Set of Postulates for the Foundation of Logic,'' *The Annals of Mathematics*, vol. 33, no. 2, pp. 346--366, Apr. 1932, articleType: primary_article / Full publication date: Apr., 1932 / Copyright © 1932 Annals of Mathematics. [Online]. Available: http://www.jstor.org/stable/1968337

[2] A. M. Turing, ``Computability and λ-definability,'' *Journal of Symbolic Logic*, vol. 2, pp. 153--163, 12 1937. [Online]. Available: http://journals.cambridge.org/article_S002248120004072X

[3] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler, ``A history of haskell: Being lazy with class,'' in *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*. ACM Press, 2007, pp. 1--55.

[4] ``The Swift programming language.'' [Online]. Available: https://developer.apple.com/library/mac/documentation/Swift/Conceptual/Swift_Programming_Language/index.html

[5] ``Using Swift with Cocoa and Objective-C.'' [Online]. Available: https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/BuildingCocoaApps/

[6] R. Hindley, ``The Principal Type-Scheme of an Object in Combinatory Logic,'' *Transactions of the American Mathematical Society*, vol. 146, pp. 29--60, 1969. [Online]. Available: http://dx.doi.org/10.2307/1995158

[7] R. Milner, ``A theory of type polymorphism in programming,'' *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348--375, 1978.

[8] ``New playgrounds part 2 - sources,'' https://developer.apple.com/swift/blog/ ?id=26.

[9] ``Discriminated Unions in F#,'' https://msdn.microsoft.com/en-us/library/ dd233226.aspx.

[10] C. Okasaki, *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press, 1998.

[11] R. Milner, L. Morris, and M. Newey, *A Logic for Computable Functions with Reflexive and Polymorphic Types*. IRIA-Laboria, 1975, pp. 371--394.

[12] ``Chris Lattner on choosing between methods and global functions in Swift,'' https://devforums.apple.com/message/1074064#1074064.

[13] C. Eidhof, F. Kugler, and W. Swierstra, *Functional Programming in Swift*, 1st ed. objc.io, 2014.

[14] E. Moggi, ``Notions of computation and monads,'' *Information and Computation*, vol. 93, no. 1, pp. 55 -- 92, 1991, selections from 1989 {IEEE} Symposium on Logic in Computer Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540191900524

[15] S. L. Peyton Jones and P. Wadler, ``Imperative functional programming,'' in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 71--84. [Online]. Available: http://doi.acm.org/10.1145/158511.158524

[16] J. Launchbury and S. L. Peyton Jones, ``State in haskell,'' *Lisp Symb. Comput.*, vol. 8, no. 4, pp. 293--341, Dec. 1995. [Online]. Available: http: //dx.doi.org/10.1007/BF01018827

[17] P. Wadler, ``Monads for functional programming,'' in *Advanced Functional Programming*, ser. Lecture Notes in Computer Science, J. Jeuring and E. Meijer, Eds. Springer Berlin Heidelberg, 1995, vol. 925, pp. 24--52. [Online]. Available: http://dx.doi.org/10.1007/3-540-59451-5_2 [18] ``Scala language,'' http://www.scala-lang.org.

[19] ``F# language,'' http://fsharp.org.

[20] S. Lane, *Categories for the Working Mathematician*, ser. Graduate Texts in Mathematics. Springer New York, 1998. [Online]. Available: http://books. google.rs/books?id=eBvhyc4z8HQC

[21] ``Haskell Functor,'' https://www.haskell.org/haskellwiki/Functor.

[22] ``Haskell Monad,'' https://www.haskell.org/haskellwiki/Monad.

[23] C. Mcbride and R. Paterson, ``Applicative programming with effects,'' *J. Funct. Program.*, vol. 18, no. 1, pp. 1--13, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1017/S0956796807006326

[24] ``Haskell Applicative Functor,'' http://hackage.haskell.org/package/base-4.7.0.2/docs/Control-Applicative.html.

[25] T. Petricek and D. Syme, ``The F# computation expression zoo,'' in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, M. Flatt and H.-F. Guo, Eds. Springer International Publishing, 2014, vol. 8324, pp. 33--48. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04132-2_ 3

[26] ``Introduction to error handling programming guide for Cocoa,'' 2011. [Online]. Available: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ErrorHandlingCocoa/ErrorHandling/ErrorHandling.html

[27] P. Wadler, ``Comprehending monads,'' in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90. New York, NY, USA: ACM, 1990, pp. 61--78. [Online]. Available: http://doi.acm.org/10.1145/ 91556.91592