Имплементација монада у програмском језику Swift 1.1

Ивица Миловановић

Садржај **– Монаде су моћан и ефективан алат из арсенала функционалног програмирања. У овом раду ћемо показати како се ове структуре, настале у оквиру математичке теорије категорија, а популаризоване кроз програмски језик Haskell, могу имплементирати и користити у програмском језику Swift, којим се израђују апликације за OS X и једну од две најпопуларније мобилне платформе, iOS.**

Кључне речи **– Haskell, монаде, Swift**

I. Увод

Монаде су апстрактне структуре настале у оквиру теорије категорија. Значај монада за програмирање описао је Моџи у свом раду [1], а у практичну употребу су ушле најпре кроз програмски језик Haskell [2], [3], [4]. 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,

И. Миловановић, Рачунарски факултет универзитета Унион, Београд, Србија (e-mail: imilovanovic12@raf.edu.rs).

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) [5], и F# (.NET платформа) [6]. Последњи у низу модерних језика, чије су многе одлике инспирисане функционалном парадигмом, је Swift [7], [8], настао под окриљем компаније Apple, ради развоја апликација за OS X и iOS платформе. Језик се може користити за објектно-орјентисани развој софтвера, користећи Cocoa и Cocoa Touch оквире [9]. Поред тога, статички типови, инференција типа, функције првог реда, ламбда изрази, могућност дефинисања додатних оператора, генерички типови итд. су неке од одлика које овај језик чине веома погодним за писање кода у функционалном стилу. С обзиром на то да тренутно постоји преко милион апликација написаних за iOS, а да је Swift језик будућности за ову платформу, изузетна је прилика да се функционално програмирање промовише као парадигма која у многим ситуацијама може да се искористи за писање краћег, јаснијег, прегледнијег и безбеднијег кода. У овом раду показујемо како се концепт монаде може искористити за побољшавање свакодневног рада са неколико најкоришћенијих типова у Swift језику.

A. Дефиниција

Појмови функтора и монаде прецизно и формално су дефинисани у теорији категорија [10]. Ако су C и D категорије, тада функтор F сваком $X \in A$ придружује објекат $F(X) \in B$ и сваком морфизму $f : X \to Y \in A$ придружује морфизам $F(f) : F(X) \to F(Y) \in B$. Сваки функтор мора да очува морфизме идентитета и композиције. Ендофунктор је функтор који дефинише пресликавање категорије у себе саму. Монада је ендофунктор $F : C \to C$, заједно са две придружене природне операције, $\eta : I \to F$, где је I индентитет фунтор у категорији C, и $\mu : F \circ F \to F$. Свака монада мора да задовољава леви и десни идентитет и асоцијативност.

B. Имплементација у програмском језику Haskell

Haskell имплементира функтор и монаду као класе типова [11], [12].

```
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. Оператор `>>` 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 [13] и за такве случаје су предложили апстракцију апликативног функтора, која је у језику Haskell такође имплементирана као класа типова [14]:

```
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.

C. 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) [15]. 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 [15].

II. Монаде у језику Swift

A. Детаљи имплементације

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

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

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 `>>=` оператору и F# `Bind` функцији и метод `bind<U>(m:M<U>)->M<U>`, аналоган Haskell `>>` 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.

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.

B. Глобалне функције, оператори и методи

Важно техничко питање које се поставља приликом имплементације монада 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)`). 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. Функција `flatten` мора бити глобална јер тренутно није могуће специјализовати методе генеричких типова. Из истог разлога смо користили оператор `<*>` уместо одговарајућег метода. Креатор Swift језика, Крис Латнер, наговестио је на званичном развојном форуму компаније Apple да би се ово могло променити у будућим верзијама компајлера [16] и његова изјава подупире нашу одлуку да изаберемо методе када год је то могуће.

C. 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` 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.

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

`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 $f : T \to U$ на садржај `Optional<T>` и враћа резултат као `Optional<U>`. Функција `pure` је тривијална - узима дату вредност и враћа је 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. 1.а.

```
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)

Сл. 1. (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`

Без `bind` метода, решавање сличног проблема, у тренутној верзији језика, захтева коришћење угњеждених `if-let` блокова. Додавање сваке следеће функције у ланац `if-let` везивања, захтева додатно угњеждење, што врло брзо постаје потпуно нечитљиво, док приликом коришћења `bind` метода читљивост остаје 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. 1.b). Слично `bind` методу, предност апликативног стила нарочито долази до изражаја када функција, која се примењује на опционе вредности, има много аргумената.

D. 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 [17]. 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, запакован у `Result<T>`. Монадска операција `pure` је тривијална - узима прослеђену вредност `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. 1.a), може се имплементирати тако да, уместо опционог резултата, враћа `Result<T>` (Сл. 2.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))                    (5 times)
210         }
211         return .Error(OperationNotFound)
212 }
213
214 let result: Result<Double> =                                                  (Enum Value)
215 perform ("+") (2)(3)
216     .bind { result in perform ("*") (result)(3) }                             (Enum Value)
217     .bind { result in perform ("/") (result)(2) }                             (Enum Value)
218
219 println(result)                                                               "Result(7.5)"
```

(a)

```
221 let greater: Result<Double> = max <^> perform ("/") (5)(2) <*> perform ("-") (10)(7)   (Enum Value)
222 println(greater)                                                                       "Result(3.0)"
```

(b)

Сл. 2. (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. 2. 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. 1.b, може се користити са `Result<T>` на потпуно исти начин као са `Optional<T>` (Сл. 2.b).

E. 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. 3. 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)

Сл. 3. (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

III. Закључак

Резултати овог рада показују погодност Swift језика за једноставну имплементацију и коришћење монада. 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 пожељне 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. Овим се, такође, појављује занимљив практични и истраживачки проблем за садашње софтверске фирме у којима функционално програмирање није више само академска творевина, већ ефикасно средство највећих комерцијалних развојних плаформи.

Литература

[1] 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

[2] 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

[3] 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

[4] 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

[5] "Scala language," http://www.scala-lang.org.

[6] "F# language," http://fsharp.org.

[7] "Swift language," http://www.apple.com/swift/.

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

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

[10] 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

[11] "Haskell Functor," https://www.haskell.org/haskellwiki/Functor.

[12] "Haskell Monad," https://www.haskell.org/haskellwiki/Monad.

[13] 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

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

[15] 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

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

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

Abstract

Monads are powerful constructs used in functional programming for managing processes with side effects, but also for modelling and building complicated computations. Monads are theoretically founded in Category Theory and widely popularised by the Haskell programming language. In this paper, we show how monads can be implemented and used in Swift, the new language for developing iOS and OS X applications.

MONADS IN SWIFT

Ivica Milovanović