Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/09BE17Novakovic

TEORIJA KATEGORIJA KAO OSNOVA ZA RAZVOJ POSLOVNIH VEB APLIKACIJA

CATEGORY THEORY AS A BASIS FOR DEVELOPING BUSINESS WEB APPLICATIONS

Aleksandar Novaković, *Fakultet tehničkih nauka, Novi Sad*

Oblast – SOFTVERSKO INŽENJERSTVO I INFORMACIONE TEHNOLOGIJE

Kratak sadržaj – *Cilj ovog rada jeste da pokaže kako se striktni matematički formalizmi mogu iskoristiti kao osnova za razvoj produkcionih poslovnih veb aplikacija. Shodno tome, prvi deo rada, naslovljen teorijske osnove, daje pregled oblasti fundamenata matematike, konkretno teorije kategorija. Drugi deo rada, pregled stanja u oblasti, pokazuje kako se pojmovi teorije kategorija prevode u iskaze programskih jezika. Primeri su dati u programskom jeziku Scala, pre svega zbog široke upotrebe ovog programskog jezika. Treći deo rada je prikaz slučaja u kom je razvijena ilustrativna poslovna veb aplikacija namenjena testiranju učenika.*

Ključne reči: *funkcionalno programiranje, teorija kategorija, funktor, monada, veb aplikacija, Scala*

Abstract – *Purpose of this paper is to show how one can use strict mathematical formalisms as a base for developing production ready business web applications. Consequently, first part of this paper, titled theoretical foundations, goes through parts of fundamental mathematics, specificaly category theory. Second part of the paper, overview of the state of the field, shows how category theory terms translate to expressions of programming languages. Examples are presented in programming language Scala, primarily because of it's popularity. Third part of this paper presents use case in which ilustrative business web application for student examination is developed.*

Keywords: *functional programming, category theory, functor, monad, web application, Scala*

1. UVOD

Poslednjih nekoliko godina sve popularnija paradigma programiranja postaje funkcionalna paradigma. I pored ovoga, često se tokom diskusija navode tvrdnje da funkcionalni jezici još uvek nisu primenljivi u produkciji i da im je namena isključivo akademska. Kada se primenjuje u tradicionalnim objektno-orijentisanim programskim jezicima, uglavnom se primenjuje samo mali podskup koncepata funkcionalnog programiranja.

Cilj ovog rada je da pokaže kako se striktni matematički formalizmi mogu iskoristiti kao osnova za razvoj produkcionih poslovnih veb aplikacija.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Milan Segedinac, vanr. prof.

Shodno tome, prvi deo rada, naslovljen teorijske osnove, daje pregled oblasti fundamenata matematike, konkretno teorije kategorija. Drugi deo, pregled stanja u oblasti, rada pokazuje kako se pojmovi teorije kategorija prevode u iskaze programskih jezika. Primeri su dati u programskom jeziku Scala, pre svega zbog široke upotrebe ovog programskog jezika. Treći deo rada je prikaz slučaja u kom je razvijena ilustrativna poslovna veb aplikacija namenjena testiranju učenika.

2. TEORIJSKE OSNOVE

2.1. Funkcionalno programiranje

Prvi i osnovni koncept na kom se zasniva funkcionalno programiranje je ideja da je ceo naš program pisan isključivo u čistim funkcijama (pure functions). Čista funkcija je funkcija koja ne komunicira sa spoljnim svetom, što znači da:

- totalna, a ne parcijalna, što znači da funkcija mora imati definisanu povratnu vrednost za svaku moguću ulaznu vrednost iz zadatog domena,
- deterministička, odnosno za istu ulaznu vrednost uvek vraća istu izlaznu vrednost,
- čista, nema sporednih efekata (nema komunikacije sa spoljnih svetom),
- ne menja stanje sistema, niti varijabli,
- nema *null* vrednosti,
- ne koristi refleksiju (samim tim nema ni magičnih operacija, cela logika funkcije je transparentna),
- ne baca izuzetke, što znači da se logika funkcije ne može prekinuti u bilo kom trenutku (greške se vraćaju kao deo povratne vrednosti funkcije) [1].

2.2. Teorija kategorija

Veliki značaj za funkcionalno programiranje ima *teorija kategorija*. Teorija kategorija formalizuje matematičke strukture i njene koncepte u pogledu označenih usmerenih grafova zvanih *kategorije*.

Kategorija je uređena četvorka **A** = *(O, hom, id, ∘)* gde su:

1. klasa *O*, čiji se članovi nazivaju **A**-objektima,
2. za svaki par **A**-objekata *(A, B)*, klasa *hom(A, B)*, čiji članovi se zovu **A**-morfizmi od *A* do *B*,
3. za svaki **A**-objekat *A*, morfizam id_A: $A \rightarrow A$, se naziva **A**-identitet nad *A*,
4. zakon kompozicije koji vezuje za svaki **A**-morfizam $f : A \rightarrow B$ i svaki **A**-morfizam $g : B \rightarrow C$ **A**-morfizam $g \circ f : A \rightarrow C$, koji se naziva kompozicija *f* i *g* [2].

Ako su **A** i **B** kategorije, onda je funktor od *A* ka *B* funkcija koja dodeljuje svakom **A**-objeku *A* **B**-objekat *F(A)*, i svakom **A**-morfizmu $f : A \rightarrow A'$, gde je A' **A**-

objekat, dodeljuje **B**-morfizam na takav način da važe sledeći zakoni:

1. *F* čuva kompoziciju, što znači da kad god je $f \circ g$ definisano, važi i $F(f \circ g) = F(f) \circ F(g)$,
2. *F* očuvava identični morfizam, što znači da za svaki **A**-objekat *A* važi da je $F(id_A) = id_{F(A)}$ [2].

Monada nad kategorijom **X** je uređena trojka $T = (T, \eta, \mu)$ koju čine funktor $T : X \to X$ i prirodne transformacije $\eta : id_X \to T$ i $\mu : T \circ T \to T$ takve da dva dijagrama prikazana na slici 1 komutiraju [2].

Slika 1: Komutirajući dijagrami koji predstavljaju asocijativni zakon (levi dijagram), kao i zakone levog i desnog identiteta (desni dijagram) [2]

2.3. Scala

Autor programskog jezika Scala, Martin Oderski, je započeo razvoj jezika 2001. Prva zvanična verzija je izdata 20.01.2004. godine. Na samom početku, Scala je razvijana u cilju rešavanja najčešćih problema i kritika koje je dobijao programski jezik Java. Kada se kompajlira, Scala kod se prevodi u Java bajt kod, čime ima punu interoperabilnost sa Javom i njenim bibliotekama. Kako se kompajlira u Java bajt kod, Scala program se izvršava na Java virtuelnoj mašini (JVM) [3].

Danas Scala pripada kategoriji hibridnih jezika, što znači da pokušava da objedini funkcionalne i objektno-orijentisane koncepte. Objektno-orijentisani jezik je zato što je svaka vrednost u Scali objekat, a funkcionalan je zato što je svaka funkcija vrednost.

3. PREGLED STANJA U OBLASTI

3.1. Klase tipova

Klasa tipova je abstraktna struktura koja sadrži skup funkcija koje mogu imati više različitih implementacija zavisno od tipa/instance koji im je dat. Najčešće se koristi kako bi jasno razdvojila podatke od implementacije funkcija interfejsa. Ovim dobijamo mogućnost lakog proširenja funkcionalnost struktura u bibliotekama čijem internom kodu nemamo pristup.
Svaka *klasa tipova* ima četiri važna elementa:

- definicija same klase tipova koja uključuje deklaraciju metoda koje data klasa tipova obuhvata,
- instance klase tipova,
- interfejs klase tipova koji će klijenti da koriste,
- zakoni koje klasa tipova, odnosno njene instance moraju da zadovolje [4].

3.2. Funktor

Videli smo da je funktor preslikavanje jedne kategorije na drugu kategoriju sa očuvanjem morfizama. U Scali imamo set kategoriju i endofuntkore. To znači da svaki tip možemo posmatrati kao objekat u set kategoriji, funkcije možemo posmatrati kao morfizme u ovoj kategoriji. Funktori su onda endofunktori, odnosno preslikavanje morfizama se vrši u okviru set kategorije (uvek na objekte koji već postoje u kategoriji).

Shodno tome, uz svaki funktor treba dati funkciju višeg reda, *map*, koja morfizme pre preslikavanja objekata preslikava na morfizme između preslikanih objekata.

Funktor je apstraktan konstrukt, tako da o *map* metodi treba da razmišljamo kao o metodi koja transformiše svaki element unutar funktora odjednom uz očuvanje strukture funktora. U nastavku dajemo definiciju klase tipova *Functor* [1]:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B):
F[B]
}
```

Kada smo uveli definiciju funktora u teoriji kategorija, naveli smo dva zakona koja moraju da važe za svaki funktor, *zakon identiteta* i *zakon kompozicije* [5].

3.3. Monada

Već smo pričali o monadama kada je bilo reči o teoriji kategorija i videli smo da je monada funktor *T* sa dve prirodne transformacije, μ i η. Pojam funktora u programiranju smo razradili u prethodnoj sekciji. Sada ostaje da vidimo koja je programerska interpretacija navedenih prirodnih transformacija.

U programskom jeziku Scala za prirodnu transformaciju μ tipično se koristi naziv *flatten*, a za *η pure*.

Prirodna transformacija μ je bila definisana kao:

$$\mu : T \square T \to T$$

Shodno tome znamo da join funkcija treba da bude:

```
def flatten[F[_], A](ffa: F[F[A]]):
F[A]
```

Dalje, monada ima i prirodnu transformaciju η:

```
η : A → T(A)
```

Shodno tome možemo definisati i *pure*:

```
def pure[F[_], A](a: A): F[A]
```

Kako smo formalnu definiciju monade uveli pomoću *pure* i *flatten* funkcija, u nastavku dajemo definiciju *Monad* klase tipova koja se primarno oslanja na implementaciju ove dve funkcije, dok je najčešće korišćena *flatMap* funkcija implementirana pomoću *map* i *flatten* funkcije.

```
trait Monad[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def flatten[A](ffa: F[F[A]]): F[A]
  def flatMap[A, B](fa: F[A])
    (f: A => F[B]): F[B] =
    flatten(map(fa)(f))
}
```

Prilikom uvođenja monade u teoriji kategorija, predstavili smo kroz komutirajuće dijagrame (slika 1) tri zakona koja svaka monada mora da zadovolji, *zakon asocijativnosti*, *zakon levog identiteta* i *zakon desnog identiteta* [5].

Scala pruža mogućnost komponovanja monada istog tipa pomoću *for komprehensije*, prevodeći je u poziv metoda *flatMap* i *map*.

3.4. Either monada

Either predstavlja produkt tipova *Right* i *Left*. *Right[A]* predstavlja prisustvo vrednosti tipa *A* i uspešno izvršen račun. *Left[E]* predstavlja prisustvo greške *E*, odnosno neuspešno izvršen račun [1].

Either je funktorialan u *A*, tako da metoda *map* transformiše vrednost tipa *A* u vrednost tipa *B*, dok *flatMap* transformiše vrednost tipa *A* u vrednost tipa *Either[E, A]*. Funkcija *pure* za primljenu vrednost *a* će samo vratiti vrednost *Right(a)*.

3.5. Reader monada

Ako imamo funkciju koja pristupa nekom read-only stanju, tu funkciju uvek možemo da zamenimo funkcijom koja read-only stanje prima kao parametar. To je funkcija *(A, E) => B*, koja prima "glavnu" vrednost tipa *A* i vrednost read-only stanja tipa *E* i vraća neku vrednost tipa *B*. Ako primenimo *currying* na ovu funkciju dobijamo *A => (E => B)*, funkciju koja prima samo "glavnu" vrednost tipa *A* i vraća novu funkciju koja prima vrednost read-only stanja tipa *E*, a vraća vrednost tipa *B*. Da bi rešili ovaj problema moramo uvesti novu monadu, *Reader* monadu [1]:

```
case class Reader[R, A](run: R => A)
```

Reader je funktorialan u *A*, tako da metoda *map* transformiše vrednost tipa *A* u vrednost tipa *B*, dok *flatMap* transformiše vrednost tipa *A* u vrednost tipa *Reader[R, B]*. Funkcija *pure* za primljenu vrednost *a* će samo vratiti vrednost *Reader(_ => a)*, odnosno, ignorisaće bilo kakvo primljeno okruženje.

3.6. IO monada

Iako ne smemo u čistim funkcijama da imamo izvršavanje sporednog efekta, ništa nas ne sprečava da u čistu funkciju stavimo opis efekta. Ukoliko ovako posmatramo efekte, ceo program postaje opis jednog velikog efekta, koji će na kraju da primi interpreter koda i izvrši ga. Postavlja se pitanje kako kreirati opis efekta bez njegovog izvršavanja u jezicima poput Scale, koji nisu lenji po difoltu? Ukoliko kod efekta smestimo u funkciju bez parametara, taj kod se neće izvršiti čim dobavimo referencu na funkciju, već samo onda kad pozivamo samu funkciju. Dakle, opis efekta možemo predstaviti funkcijom koja nema ulaznih parametara, a za rezultat vraća rezultat izvršavanja efekta, odnosno neku vrednost tipa *A*. Nazovimo ovakvu strukturu *IO*. U Scali bi *IO* mogli da implementiramo na sledeći način [1]:

```
case class IO[A](run: () => A)
```

IO je funktorialan u *A*, tako da metoda *map* transformiše vrednost tipa *A* u vrednost tipa *B*, dok *flatMap* transformiše vrednost tipa *A* u vrednost tipa *IO[B]*. Funkcija *pure* za primljenu vrednost *a* će samo vratiti vrednost *IO(_ => a)*, odnosno, samo će podići primljenu vrednost u efekat.

3.7. ZIO monada

IO monada nam daje mogućnost da enkodujemo interakciju sa spoljnim svetom, međutim, u realnom softveru nam pored toga verovatno treba da istovremeno imamo i mogućnost obrade grešaka, što nam omogučava *Either* monada, i injektovanja zavisnosti, što nam omogućava *Reader* monada. Da bi dobili sve ove funkcionalnosti, potrebno je da koristimo *Reader*, *Either* i *IO* monadu. Ovo bi bilo nepraktično, zato što bi morali da koristimo mnogo ugnježdenih transformatora i kod bi postao manje čitak u odnosu na imperativnu objekto-orjentisanu verziju.

Da bi rešili ovaj problem, uvešćemo novi tip koji predstavlja funkciju. Ova funkcija za ulazni parametar prima okruženje, a za rezultat vraća vrednost koja može biti greška, ili sama vrednost. Ukoliko navedenu strukturu nazovemo *ZIO*, ona bi na osnovu opisa izgledala ovako:

```
case class ZIO[R, E, A](run: R =>
Either[E, A])
```

Predstavljena struktura sadrži tri parametra tipa:

- *R* je tip okruženja od kog račun u *ZIO* monadi zavisi,
- *E* je tip greške koji račun u *ZIO* monadi može da vrati,
- *A* je tip vrednosti koju račun u *ZIO* monadi vraća [6].

ZIO je funktorialan u *A*, tako da metoda *map* transformiše vrednost tipa *A* u vrednost tipa *B*, dok *flatMap* transformiše vrednost tipa *A* u vrednost tipa *ZIO[B]*. Funkcija *pure* za primljenu vrednost *a* će samo vratiti vrednost *ZIO(_ => Right(a))*, odnosno, samo će podići primljenu vrednost u *ZIO* efekat.

4. IMPLEMENTACIJA

4.1. Specifikacija

ExamCoach je aplikacija koja služi za kreiranje i upravljanje testovima u nastavi, kao i rešavanje istih. Implementiran je server koji sadrži HTTP REST interfejs za upravljanje testovima. Sistem može da koristi bilo koja klijentska aplikacija nezavisna od tehnologije.

Sistem razlikuje tri korisničke uloge:

1. **Administrator** – administrator ima mogućnost da kreira kategorije testova koje nastavnici kasinije mogu referencirati iz svojih testova.
2. **Nastavnik** – nastavnik ima mogućnost da kreira pitanja, kao i da kombinuje pitanja u testove. Nastavnik može da pretraži kategorije testova i pokuša da nađe odgovarajuće kategorije kojim pripada njegov test. Čim se kreira, test postaje vidljiv ispitanicima koji rešavaju testove. Svaki nastavnik može da vidi samo svoja pitanja. Pored ovoga, nastavnik može da se registruje i prijavi na sistem.
3. **Učenik** – učenik ima pravo da pretražuje testove i nakon što nađe odgovarajući, da ga rešava. Test se može rešavati iz više delova i nakon što se reši, korisnik ima pravo da vidi rezultate testa i koliko je uspešno rešio test. Pored ovoga, učenik može da se registruje i prijavi na sistem.

4.2. Arhitektura

Aplikacija je orhanizovana u tri odvojena sloja: transportni sloj, servisni sloj i sloj podataka. Transportni sloj direktno zavisi od sloja servisa. Kao što mu i ime sugeriše transportni sloj veši validaciju zahteva i prosleđuje zahteve servisnom sloju. Servisni sloj je nezavisan od transportnog sloja, ali zato zavisi od sloja podataka. U servisnom sloju se nalazi cela posnovna logika sistema. Sloj podataka je nezavisan od svih ostalih slojeva. On sadrži specifikaciju interakcije našeg servisa sa bazom podataka, kao i specifikaciju upita i instrukcija koje će se izvršavati.

Ceo sistem radi sa pet entiteta i svaki entitet ima svoju komponentu u svakom od navedenih slojeva. Ti entiteti su: korisnik, kategorija, test, pitanje i instanca rešavanja testa.

Ceo projekat je podeljen u pakete i module. Moduli predstavljaju male, jasno definisane logičke celine koje se na jednostavan način mogu međusobno kombinovati. U servisnom sloju postoje po dva modula za svaki entitet koji smo naveli, jedan modul za servisnu logiku, i jedan modul za pristup repozitorijumu. Pored ovih, postoje i moduli za upravljanje heš vrednostima, generisanje jedinstvenog identifikatora i generisanje autorizacionog tokena.

4.3. Implementacija

Za implementaciju transportnog sloja je korišćena http4s funkcionalna biblioteka. Rutiranje HTTP zahteva je implementirano pomoću uklapanja oblika (*pattern matching*). Funkcije za validaciju pojedinačnih polja za rezultat vraćaju vrednost tipa *Either*. Da bi validirali kompleksnije entitete, komponovali smo više različitih funkcija kroz *for komprehensiju*, čime smo dobili kod nalik imperativnom, iako on to nije. Za predstavljanje validacionih grešaka korišćen je koprodukt tip nad kojim se vrši uklapanje tipova. Nakon validacije se pozivaju metode servisnog sloja. Kako svaka metoda servisnog sloja ima sporednih efekata (npr. komunikacija sa bazom), za povratnu vrednost se uvek vraća *ZIO* monada. Svaka metoda je implementirana kao kompozicija *ZIO* monada kroz *for komprehensiju*. Izmena vrednosti instance entiteta se obavlja tako što se kreira kopija postojeće instance uz odgovarajuće izmene polja.

Za implementaciju sloja podataka je korišćena *doobie* biblioteka koja nam daje funkcionalan API za pristup SQL bazama podataka. Nakon injektovanja vrednosti u SQL upit, vrši se kreiranje *ConnectionIO* efekata koji predstavlja opis pristupa bazi. Kada je potrebno izvršiti više upita unutar jedne transakcije, vršit se kompozicija navedenih efekata pomoću *for komprehensije*, čime dobijamo potpunu kontrolu nad izvršavanjem transakcije.

Pored implementacije modula, napisani su i testovi za iste. Svaki test ima definisan objekat koji u sebi nosi definiciju alokacije stanja sistema, kao i definiciju dealokacije stanja sistema. Stanje sistema nije deljeno među testovima, što znači da možemo paralelizovati postojeće testove na bezbedan način (ne može doći do *data race-a*). Za implementaciju testova je korišćen *ZIO-test okvir*.

5. ZAKLJUČAK

U ovom radu su predstavljeni osnovni koncepti funkcionalnog programiranja i mogućnost njihove primene u realnim situacijama. U prvom delu je izložen sažetak osnovnih koncepata teorije kategorija i istorije razvoja programskog jezika Scala. Nakon toga, je dat pregled osnovnih funkcionalnih koncepata u programskom jeziku Scala, kao i pregled značajnijih biblioteka poput *ZIO*. Treći deo daje specifikaciju projekta za upravljanje testovima i specifikaciju arhitekture projekta. Pored toga, predstavljeni su i delovi implementacije u cilju demonstracije primene funkcionalnih koncepata i pomenutih biblioteka.

Rešenje je implementirano kao serverska aplikacija kako bi omogućila korišćenje implementirane logike nezavisno od korišćene tehnologije. ZIO je korišćen za opis efekata koje server treba da izvrši i za testiranje istih. Za interakciju sa spoljnim svetom, primenjena je *http4s* biblioteka, dok je za komunikaciju sa bazom podataka korišćen *doobie*.

6. LITERATURA

[1] Chiusano, P., & Bjarnason, R. (2014). Functional programming in Scala. Manning Publications Co.

[2] Adámek, J., Herrlich, H., & Strecker, G. E. (2004). Abstract and concrete categories. The joy of cats.

[3] Odersky, M., Altherr, P., Cremet, V., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., & Zenger, M. (2006). An Overview of the Scala Programming Language Second Edition.

[4] Noel, W., & Dave, G. (2017). Scala with Cats.

[5] Lipovaca, M. (2011). Learn you a haskell for great good!: a beginner's guide.

[6] „ZIO.dev. Summary." URL: https://zio.dev/docs/overview/overview_index (pristupljeno u februaru 2020.)

Kratka biografija:

Aleksandar Novaković rođen je 12.03.1994. godine u Novom Sadu. Godine 2009. završio je Osnovnu školu „Sveti Sava" u Zvorniku. Gimnaziju „Filip Višnjić" u Bijeljini, završio je 2013. godine. Iste godine upisao je Fakultet tehničkih nauka na Univerzitetu u Novom Sadu, smer Softversko inženjerstvo i informacione tehnologije. Zvanje diplomirani inženjer elektrotehnike i računarstva stekao je 2017. godine. Iste godine upisao je master akademske studije na smeru Softversko inženjerstvo i informacione tehnologije. Položio je sve ispite propisane planom i programom.
kontakt: anovakovic01@gmail.com