Primjena optimizacijskih algoritama i strojnog učenja u kontekstu funkcijskog programskog jezika Haskell

Luka Hadžiegrić

Zagreb, Veljača 2018

“Pod punom odgovornošću pisano potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada.”

U Zagrebu, 20.2.2018

Predgovor

Ovim putem želim se zahvaliti svome mentoru dr.sc. Goranu Klepcu koji me je uveo u svijet podatkovne znanosti i pružio mi priliku da radim na ovom projektu, svojoj obitelji na velikoj potpori tijekom studiranja, te svim svojim učiteljima i profesorima koji su me vodili kroz moje dosadašnje obrazovanje, pogotovo dr.sc Janu Šnajderu.

Sažetak

Porastom kompleksnosti programskih rješenja funkcijska paradigma postaje sve značajnija u razvoju softvera jer obećava veću modularnost, stabilnost i razumljivost programskog koda.

Ovaj rad istražuje te prednosti kroz izradu Internet aplikacije i primjenu algoritama strojnog učenja i optimizacije.

Ključne riječi: haskell, optimizacija, strojno učenje, postgresql, internet aplikacija, obrada podataka

Sadržaj

Poglavlje 1

Uvod

Cilj ovog rada je istražiti primjenu čistog funkcijskog programskog jezika Haskell na praktičnom primjeru izrade pametne kuharice koja koristi metode strojnog učenja i optimizacije kako bi korisniku preporučila potencijalno zanimljiv recept ili stvorila tjedni jelovnik koji maksimalno iskorištava sastojke.

Ideja je proizašla iz nezadovoljstva nakon višegodišnjeg korištenja objektno orijentiranih programskih jezika te pojavom elemenata funkcijskog programiranja u popularnim programskim jezicima poput JavaScripta, C#a i Jave.

Današnje interakcije ljudi i različitih područja ljudskog djelovanja postaju sve složenije te važnost računarske znanosti i računarstva kao glavnog vezivnog tkiva i kanala za razmjenu i obradu informacija postaje sve veća.

Iz tog razloga pojavljuju se sve složeniji sustavi i primjene programskih rješenja nad kojima se mora brzo iterirati uz maksimalnu sigurnost i kvalitetu. Takva dinamika zahtjeva efektivne alate za upravljanje kompleksnošću i tu se objektno orijentirani pristup pokazuje sve manje poželjnim izborom, dok se funkcijski pristup čini sve pogodnijim.

Ovaj trend korištenja funkcijskih programskih jezika prati sve veći broj manjih firmi koje nemaju teret zastarjelog (eng. *legacy*) koda, ali ne zaostaju niti veće firme poput FaceBook-a koji je razvio svoj sustav Sigma[6] za borbu protiv neželjenih poruka (eng. *spam*) u Haskellu.

Poglavlje 2

Haskell

Slika 2.1: Haskell logo

Haskell je čisto funkcijski programski jezik te ima sve odlike koje se mogu pronaći kod takvih jezika, kao što su čistoća (eng. *purity*), lijenost (eng. *lazyness*), referencijalna transparentnost (eng. *referential trnsparency*), funkcije višeg reda (eng. *higher order functions*), nepromjenjive podatke (immutable data. ,) ne striktnu semantiku (eng. *nonstrict semantics*), sporedne efekte (eng. *side effects*) i još mnoge druge.[3]

Sve to čini Haskell vrlo moćnim i ekspresivnim programskim jezikom koji omogućava da jasno i jednostavno izrazimo koncepte koji nisu lako izvedivi u klasičnim imperativnim jezicima.

Lijenost i ne striktna semantika nam omogućavaju da jednostavno baratamo sa vrlo apstraktnim podacima kao što su beskonačne liste ili *bottom* vrijednosti. Čistoća, referencijalna transparentnost i nepromjenjivost podataka osigurava lako razumijevanje koda te njegovo konzistentno ponašanje dok nam funkcije višeg reda daju višestruku iskoristivost i kompozitnost koda (eng. *code reusability*).

Čistoća je važno svojstvo Haskella a svodi se na to da funkcija uvijek vraća vrijednost i ne može imati nikakve sporedne efekte, kao recimo mijenjanje globalnog stanja. Ovo je vrlo bitno jer olakšava mnogo stvari kao što je na primjer konkurentnost koda gdje možemo biti sigurni da dva odvojena procesa neće pokušati pisati u istu varijablu.

Ipak, ponekad želimo imati sporedne efekte u našim funkcijama (takozvani prljavi ili efektni kod). Želimo da naša funkcija, osim što vraća vrijednost piše i u bazu podataka, ili lansira nuklearne rakete. U Haskellu je i takav kod u principu čist jer se izvodi preko monadskog sučelja kroz koje su takve akcije *eksplicitno* definirane te se iz samog potpisa funkcije vidi da li ona izvodi nekakvu potencijalno "prljavu" operaciju, stanje se npr. eksplicitno definira i provlači kroz daljnje funkcije skriveno iza monadskog sučelja koje nam daje "imperativni" osjećaj uz garanciju čistoće.[1]

2.1 Osnovni koncepti

Pošto se funkcijski stil u mnogočemu razlikuje od imperativnog stila programiranja, bitno je upoznati se sa osnovnim konceptima kako bi ostatak rada bio razumljiv. Monade su već bile spomenute ali uz njih još postoje i transformatori monada (eng. *monad transformers*), aplikativni funktor (eng. *applicative functor*), funktor (eng. *functor*), monoid (eng. *monoid*), polugrupa (semigroup. ,) klase tipova (eng. *type class*), algebarski podatkovni tipovi (eng. *algebraic data types*) i drugo.

2.1.1 Funkcije

Funkcije su osnovni gradivni elementi u Haskellu te se ponašaju kao i sve druge vrijednosti. Pogledajmo primjer definicije funkcije koja prima dva slova (`Char`), pretvara ih u brojčanu reprezentaciju te vraća zbroj:

```
1  f :: Char -> Char -> Int
2  f a b = ord a + ord b
```

Listing 1: Definicija funkcije

Prva linija je tipski potpis (eng. *type signature*) funkcije koji nam govori koje tipove podataka funkcija prima i koji tip podatka vraća. Na drugoj liniji je implementacija te funkcije. Kao što se može primijetiti kod funkcije `ord` čiji je tipski potpis `ord :: Char -> Int`, kod primjene funkcije se ne koriste zagrade.

Ovakav način primjene funkcije je koristan zbog koncepta zvan *currying* koji u Haskellu znači da funkcija uvijek prima samo jedan argument. Funkcije koje izgledaju kao da primaju više argumenata zapravo primaju samo jedan argument te kao rezultat vraćaju novu funkciju koja opet očekuje idući argument. To se zove djelomična primjena (eng. *partial application*) te je vrlo korisna za brzu izradu novih prilagođenih funkcija. Kada bi dali samo jedan argument našoj funkciji `f` npr. `f 3` dobili bi novu funkciju čiji bi potpis bio `Char -> Int` a neka zamišljena implementacija bi bila `f' b = ord 3 + ord b`.

2.1.2 Algebarski podatkovni tipovi

Algebarski podatkovni tip je vrsta kompozitnog tipa podatka. Dvije uobičajene varijante su produktni tip (eng. *product type*) i sumarni tip (eng. *sum type*). Pogledajmo primjer definicije jednostavne ulančane liste:

```
1  data List a
2    = Nil
3    | Cons a (List a)
```

Listing 2: Definicija ulančane liste

`List` je konstruktor tipova (eng. *type constructor*) sa tipskom varijablom `a` (eng. *type variable*) te se može koristiti samo u potpisu funkcije. Da bi postao konkretni tip moramo mu dati konkretnu vrijednost na mjesto `a`, npr. `Int`. Iz tog razloga se smatra i funkcijom tipske razine (eng. *type level function*).

U svojoj definiciji `List` daje dva podatkovna konstruktora `Nil` i `Cons a (List a)`. `Cons` se može smatrati funkcijom sa potpisom `a -> List a -> List a`. Kao što možemo primijetiti iz primjera, definicija tipa može biti rekurzivna.

Pogledajmo primjer izrade liste cijelih brojeva:

```
1  list :: List Int
2  list = 1 `Cons` 2 `Cons` 3 `Cons` Nil
```

Listing 3: Deklaracija liste brojeva

Vrlo praktična značajka jezika je infix notacija koja nam omogućava da prvi argument funkcije damo prije samog imena funkcije tako da ga okružimo sa znakom "`".

2.1.3 Klase tipova

Klase tipova su slične sučeljima (eng. *interfaces*) iz objektno orijentiranih jezika a imaju i elemente klasa. Glavna razlika je da se definiraju odvojeno od podatkovnog tipa što pruža daleko veću fleksibilnost.

Pogledajmo primjer definicije tipske klase `FooBar` koja definira dvije "metode" `bar` i `foo` te instancu te klase za `Int` podatkovni tip:

```
1  class Num a => FooBar a where
2    foo :: a -> a -> a
3    bar :: a -> a -> a
4
5  instance FooBar Int where
6    foo a b = a + b
7    bar a b = a - b
```

Listing 4: Deklaracija i instanca tipske klase

U ovom slučaju klasa `FooBar` zahtijeva da tip `a` nad kojim se definira sučelje ima prethodno implementiranu klasu `Num` jer se koriste operacije zbrajanja i oduzimanja koje su definirane u toj klasi. Također, bitno je primijetiti razliku između tipske varijable `a` i argumenta funkcije `a`.

2.1.4 Teorija kategorija

Velik dio koncepata u Haskellu proizlazi iz teorije kategorija (eng. *category theory*) tako da je bitno razumjeti neke osnovne pojmove kao što su *polugrupa*, *monoid*, *funktor*, *aplikativ*, *monada* i *transformator monade*.

2.1.4.1 Polugrupa

Polugrupa je svaki tip podatka nad kojim se može implementirati klasa `Semigroup` koja ima slijedeću definiciju:

```
1  class Semigroup a where
2    (<>) :: a -> a -> a
3    ...
```

Listing 5: Definicija polugrupe

Klasa definira još neke funkcije ali izdvojen je samo operator <> (funkcije koje u nazivu sadrže samo znakove automatski postaju infix operatori). Polugrupa je u principu samo sučelje koje definira binarnu operaciju.

Zakoni

Većina osnovnih klasa mora zadovoljavati određena pravila. Ta pravila nam govore puno o svojstvima koja imaju članovi te klase i omogućavaju nam donošenje dodatnih zaključaka o ponašanju našeg koda. Polugrupa konkretno mora zadovoljavati samo jedan zakon. Zakon asocijativnosti.

```
(x <> y) <> z = x <> (y <> z)
```

Listing 6: Zakoni polugrupe

Jedan primjer gdje nam je ovaj zakon koristan je kad imamo listu podataka koji spadaju u polugrupu te želimo provesti neku binarnu operaciju nad njima. Budući da znamo da zadovoljavaju svojstvo asocijativnosti onda sa sigurnošću znamo da možemo rascijepati tu listu na dijelove i paralelizirati tu operaciju bez opasnosti od grešaka.

2.1.4.2 Monoid

Monoid je "nastavak" na polugrupu te samo dodaje *neutralan* element u definiciju.

```
1  class Semigroup a => Monoid a where
2    mempty :: a
3    ...
```

Listing 7: Definicija monoida

Zakoni

```
                    mempty <> x = x
                    x <> mempty = x
              (x <> y) <> z = x <> (y <> z)
```

Listing 8: Zakoni monoida

2.1.4.3 Funktor

Funktor klasa definira operaciju mapiranja na neki podatkovni tip, odnosno omogućava nam da primijenimo neku operaciju na sadržaj neke strukture koja implementira ovu klasu.

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

Listing 9: Definicija funktora

Zakoni

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

Listing 10: Zakoni funktora

2.1.4.4 Aplikativni funktor

Aplikativ leži između monade i funktora te nam služi da primijenimo funkciju koja se nalazi unutar neke strukture na sadržaj neke druge strukture istog tipa.

```
1  class Functor f => Applicative f where
2    pure  :: a -> f a
3    (<*>) :: f (a -> b) -> f a -> f b
```

Listing 11: Definicija aplikativa

Kao što vidimo, aplikativ je ovisan o funktoru i definira dvije ključne funkcije. `pure` nam omogućava da *omotamo* vrijednost u strukturu, dok nam `<*>` daje sposobnost da primijenimo funkciju iz jedne strukture na sadržaj druge istog tipa.

Zakoni

```
pure id <*> v = v
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w
```

Listing 12: Zakoni aplikativa

2.1.4.5 Monada

Monade imaju specijalno mijesto u Haskellu, čak postoji i sintaktički šećer (eng. *syntactic shugar*) ugrađen u sam jezik (takozvana *do* sintaksa) koji nam omogućava da pišemo kod u imperativnom stilu a u pozadini zapravo koristi monadsko sučelje. U principu, monade su sučelje za povezivanje sekvencijalnih akcija.

```
1  class Applicative m => Monad m where
2    (>>=)  :: m a -> (a -> m b) -> m b
3    (>>)   :: m a -> m b -> m b
4    m >> n = m >>= \_ -> n
```

Listing 13: Definicija monade

Zakoni

```
                    pure a >>= k = k a
                      m >>= pure = m
          m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Listing 14: Zakoni monade

2.1.4.6 Transformatori monada

Transformatori monada su vrlo samo opisni. Oni nam omogućuju da "prevedemo" jednu monadu u drugu.

```
1  class MonadTrans t where
2    lift :: Monad m => m a -> t m a
```

Listing 15: Definicija transformatora monade

Zakoni

```
lift . pure = pure
lift (m >>= f) = lift m >>= (lift . f)
```

Listing 16: Zakoni transformatora monada

Bibliografija

[1] *Functional programming*. 2014. URL: https://wiki.haskell.org/Functional_programming#Purity (pogledano 30. 1. 2018).

[2] Leo Mršić Goran Klepac. *Poslovna inteligencija kroz poslovne slučajeve*. Lider press d.d., 2006. ISBN: 9539547210.

[3] *Haskell Language*. 2018. URL: https://web.archive.org/web/20180120220037/https://www.haskell.org/ (pogledano 30. 1. 2018).

[4] Paul Hudak i dr. „A history of Haskell: being lazy with class". *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM. 2007, str. 12–1.

[5] *Lazy vs non-strict*. 2016. URL: https://wiki.haskell.org/Lazy_vs._non-strict (pogledano 29. 1. 2018).

[6] Simon Marlow. *Fighting spam with Haskell*. 2015. URL: https://web.archive.org/web/20180117194306/https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/ (pogledano 10. 1. 2018).

[7] Peter Seibel. *Coders at work: Reflections on the craft of programming*. Apress, 2009.