Debagovati u Haskelu ili ne, pitanje je sad

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Vladimir Batoćanin 074/2016 vladimir@wings.rs,
Stefan Stefanović,
Jovan Ležaja, 473/2018, jovanlezaja@hotmail.com,
Đorđe Jovanović

17. april 2020

Sažetak

Ovaj seminarski rad obrađuje temu debagovanja u Haškel-u, odnosno da li je u dotičnom jeziku moguće tradicionalno debagotati i ako ne, zašto? Kakva je podrška za debagovanje u ugrađenom debageru i koliko su pouzdani drugi debageri, i da li ima smisla uopšte debagovati Haškel kod sa nečim osim GHCi-a. Osim GHCi debagera, obrađeni su Hud debager, Hat alat i Debug biblioteka. Za svaki debager su obrađeni najkorišćenije funkcije i prednosti i mane istih. Za Hud debager je obrađen i objašnjen strukturno orijentisan metod debagovanja, podržanost u različitim kompajlerima i rasprostranjenost.Dok je za Het alat i Debug biblioteku obrađeno generisanje i analiza traga, kao i pregled interaktivih alata koji na različite načine analiziraju program i pomažu programeru pri debagovanju. Za potpuno razumevanja rada je predviđeno barem bazično predznanje Haskela.

Sadržaj

1 Uvod

Softverski bag se može definisati kao bilo koje nepredviđeno ponašanje programa. Popravljanje programa u kome su prisutni bagovi se sastoji iz sistematičnog proveravanja ponašanja programa, tj. da li dobijamo očekivani izlaz za dati ulaz. Svrha ovog procesa jeste otkrivanje scenarija u kome program ne daje ispravnu povratnu informaciju. Nakon ovog postupka najčešće sledi lociranje uzroka problema, ali je isto tako moguće da samim ispitivanjem postojanja baga implicitno dobijemo dovoljno informacija o njegovoj lokaciji. Ovaj proces se naziva **debagovanje**[8].

Pošto ovaj proces najčešće nije jednostavan, oformljeni su razni principi, metodologije i alati za debagovanje. Međutim, oni su se razvijali primarno za imperativne jezike, što znači da većina njih nije efikasna sa Haskelom, koji je čist funkcionalni jezik. Jedna od najbitnijih karakteristika programskog jezika Haskel je sam dizajn jezika, koji podstiče pisanje bezbednog i robusnog koda. Ovome najviše doprinosi stroga dinamička tipiziranost, što znači da interpretator javlja grešku istog momenta kada ne može sa sigurnošću da odredi tipove tj. potpise svih funkcija. Jos jedna vrlo bitna karakteristika Haskela je da podržava funkcije višeg reda kao i lenjo tj. nepotpuno izračunavanje. Ova karakteristika drastično smanjuje efikasnost tradicionalnih debagera[5].

Da bismo razumeli proces debagovanja Haskel programa, moramo se prvo osvrnuti na originalne osnovne principe debagovanja. Jedno od prvih i najzastupljenijih mišljenja o debagovanju je da programski kod treba dokazati kao svaki matematički iskaz, čime se eliminiše potreba za debagovanjem. Ovaj princip je iz mnogo razloga bio nepogodan jer su se programski jezici razvijali u smeru koji je njihov kôd činio sve teže dokazivim[5].

Izuzetak je strukturalni stil programiranja, koji propagira da se programi pišu u vrlo jednostavnim nezavisnim celinama. Ovaj princip je implementiran u mnogim programskim jezicima kroz procedure, funkcije i predikate. Za razliku od ostalih programskih jezika, Haskel u svojoj srži ima strukturni stil programiranja, čime se ohrabruje matematičko dokazivanje njegovog koda[5].

2 Matematičko dokazivanje

Za Haskel se veoma lako nalazi analogon na formalno matematičkom jeziku, zbog toga što strukturni dizajn implementira do na strukture podataka. Pošto su one vrlo temeljno definisane, moguće je indukcijom uopštiti program od baznog slučaja ka mnogo kompleksnijim. Ovo nam dozvoljava da već u fazi inicijalnog pisanja kôda dokažemo da je naš program matematički korektan. U ovom kontekstu se najčešće koristi metod *struktruralne indukcije* (eng. *structural induction*). Ovo je moguće isključivo zbog rekurzivno definisanih struktura podataka u Haskelu, pri čemu se koristi operator | (ili) koji u prevodu na formalni matematički jezik predstavlja uniju skupova[10]

Primer 2.1 *Rekurzivno definisanje liste u Haskelu:*

```
1000 data Lista x = PraznaLista | Cons a (Lista x)
```

Znajući ovo, vrlo lako možemo dokazati korektnost programa koji koriste liste uz pomoć matematičke indukcije, gde bi nam baza indukcije bila slučaj prazne liste, a induktivni korak rekurzivni poziv liste čijom dekonstrukcijom dobijamo izolovani element, kao i listu za koju smo pretpostavili da važi induktivna hipoteza.

Primer 2.2 *Primer rekurzivno definisane funkcije:*

```
1000 sum :: [Int] -> Int
     -- baza indukcije
1002 sum [] = 0
     -- induktivna hipoteza koja vazi za xs
1004 -- induktivni korak dodavanja jednog elementa x na xs
     sum (x:xs) = x + sum xs
```

3 GHCi Debager

GHCi debager nam omogućava da u željenim momentima zaustavimo program i proverimo vrednosti pojedinačnih promenljivih preko *tačaka zaustavljanja* (eng. *breakpoints*). Takodje vrlo bitna funkcionalnost je korak po korak izvršavanje programa sa zaustavljanjem. Izuzetak od ove funkcionalnosti su vec prekompilirane biblioteke u koje nije moguće ući u okviru međukoraka.

3.1 Tačke zaustavljanja i inspekcija varijabli

Iako je moguće zaustaviti program na bilo kom izrazu odnosno liniji radi inspekcije varijabli, nije moguće proveriti tip i vrednost varijabli koje već nisu izračunate. Ovo je posledica činjenice da se u Haskel-u ne vrši zaključivanje tipova tokom izvršavanja programa. Naravno, uvek je moguće forsirati dedukcije tipa, odnosno naterati program da nastavi izvršavanje taman toliko da usko odredi sa kojim tipom podataka se radi. Problem kod ovog pristupa se javlja u slučajevima kada bismo u bloku kôda koji treba da se izvrši da bismo dobili definitni tip željene promenljive postoji ugnježdena tačka zaustavljanja, što uništava linearnost inspekcije i debagovanja kôda.

Posledice ovog problema se mogu amortizovati uvođenjem parcijalnog izračuvanja tipa izraza, umesto izračunavanja vrednosti celog izraza. Kao i uvodjenje posebne komande za ispisivanje jos neevaluiranih vrednosti, ovo je vrlo korisno s obzirom da svaki tip pre nego što može konvencionalno da se ispisuje mora da ima implementiranu funckiju za prikazivanje (eng. *show*). Neevaluirane vrednosti Haskel rešava uvođenjem *obećanja* (eng. *thunk*), koje se uvek koriste pri lenjom izračunavanju. Nedostatak ove implementacije je to što bilo koji izraz koji se lenjo odseče i ne izračuna se do kraja (na primer desna strana izraza konjukcije gde je prvi argument*False*), što znači da ni obećanje koje se nalazilo u odsečenom delu izraza nikada neće biti evaluirano.[3]

3.2 Trace

U tradicionalnim nefunckionalnim jezicima se podrazumeva postojanje debagera, koji prati tačnu sekvencu poziva koji su se izvršavali od pokretanja programa. Ovo je izutetno teško implementirati u programskim jezicima kao što je Haskel zbog toga što se izrazi i funkcije ne izračunavaju tačnim redosledom kojim su pozvani već se izračunavaju tek kada su potrebni za neko drugo izračunavanje (eng. *demand-driven execution*). Idealno rešenje bi bilo da se programera apstrahuje evaluacija koja bi izgledala što više kao ona u imperativnom jeziku, dok bi interno bio čuvan pravi redosled poziva i evaluacija, ovo rade neki Haskel debageri, ali GHCi nema tu funkcionalnost, odnosno ne pravi leksicki stek poziva (eng. *lexical call stack*) , on nam isključivo daje da na tackama zaustavljanja imamo pregled prethodnih koraka evaluiranja.

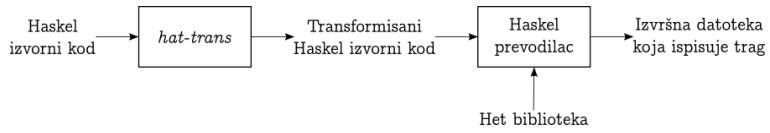
4 Debagovanje korišćenjem Heta

Het (eng. *Hat* – ***Ha**skell **t**racer*) je alat koji se koristi za generisanje *traga* (eng. *trace*) prilikom izvršavanja Haskel programa i nadziranje tako generisanog *traga* [1]. Smatra se jednim od najnaprednijih alata za debagovanje u Haskelu. Prednost alata Het u odnosu na ostale alate za debagovanje se ogleda upravo u upotrebi *traga*, jer se programeru pruža pogled unutar "crne kutije", tj. sva izračunavanja u našem Haskel programu bivaju razmotana u niz redukcija koje programer može da analizira korišćenjem različitih interaktivnih alata, pri čemu svaki od njih na različite načine interpretira generisane tragove i omogućava široki spektar analiza izračunavanja Haskel programa. Ovaj alat nije deo nekog prevodioca ili interpretatora za programski jezik Haskel, što se moglo smatrati prednošću u vremenu kada su se koristili različiti prevodioci za Haskel programski jezik, pa samo njegovo postojanje i održavanje nije bilo tesno vezano za postojanje i održavanje nekog specifičnog prevodioca, odnosno interpretatora [1]. Međutim danas, u vremenu kada je *GHC* (*GHC* – *Glasgow Haskell Compiler*) najpristupačniji Haskel prevodilac, ta osobina Heta ne mora nužno da se smatra prednošću, uzimajući u obzir da se sa konstantnim izlaskom novih verzija *GHC*-a javlja potreba za konstantnim održavanjem. U ovom radu naglasak će biti na korišćenju alata Het kao debagera, no on može da se koristi i u svrhe posmatranja kako funkcioniše korektno napisan Haskel program [11]. Nažalost, usled zastarelosti biblioteka koje koristi alat Het, autori rada nisu uspeli da osposobe alat na svojim mašinama nakon više pokušaja. Ova situacija nije začuđujuća, uzimajući u obzir da je još 2008. godine Berni Poup (eng. *Bernie Pope*) u svom radu rekao sledeće: "Het nudi nekoliko moćnih alata koji obuhvataju različite stilove debagovanja, ali je praćen negativnim uticajem na performanse, pritom nije dobro integrisan sa opšteprihvaćenim programerskim okruženjem (*GHC*), čime otežava njegovu upotrebu" [7].

Alat Het pruža programeru uvid u detalje izračunavanja pri izvršavanju Haskel programa korišćenjem *tragača* (eng. *tracer*). Koristeći se informacijama koje generiše *tragač* moguće je locirati greške u našem kôdu (ukoliko takvih ima). Sleđenje *tragova* izračunavanja u Hetu se sastoji iz dve faze: prva je *ostavljanje traga* (eng. *trace generation*), a druga je *pregledanje traga* (eng. *trace viewing*) [1].

4.1 Ostavljanje traga

U fazi *ostavljanja traga* se pokreće program koji treba da se debaguje tako da ispisuje *trag* u određenu datoteku. Da bi program ispisivao *trag* u datoteku, potrebno ga je prvo transformisati korišćenjem alata koji se sadrži u Hetu pod nazivom *hat-trans*. U tom procesu se naš Haskel program transformiše u Haskel program koji se prevodi i povezuje (eng. *linking*) sa odgovarajućim bibliotekama koje pruža Het [1]. Tako transformisan program se prevodi i pokreće, pri čemu transformisan program radi isto što i originalni program, uz dodatak da ispisuje *trag* u određenu datoteku. Grafički prikaz ostavljanja traga je prikazan na slici 1.



Slika 1: Ostavljanje traga u Hetu

Jedna od razlika koja se javlja kod Heta u odnosu na neke druge debagere je ta da je uloga transformisanja izvornog koda prepuštena računaru, tj. da je programer oslobođen od dodavanja novog koda u cilju debagovanja, kao što to imamo kod Hud (eng. *Hood*) debagera ubacivanjem *observe* ključne reči. Ta osobina alata Het se može smatrati vrlinom, obzirom da je cilj debagovanja da bude što "bezbolniji", kako po originalni izvorni kod, tako i po programera. Naime, kada je u pitanju menjanje izvornog koda u cilju debagovanja, težnja programera je da napravi što manje izmena u samom izvornom kodu, a pritom se uvodi još jedan faktor greške ukoliko se taj kod menja ručno. Takođe, prilikom pokretanja tako transformisanog programa, datoteku sa tragom je moguće koristiti neograničen broj puta, s obzirom da je ta datoteka sačuvana u sekundarnoj memoriji ne bi li više alata iz Heta moglo da se koristi datotekom u jednom pokretanju programa. Po pitanju zauzetosti memorije ova osobina nije baš poželjna jer, u slučaju debagovanja kompleksnijih programa, datoteka koja sadrži trag izvršavanja programa može da bude velika u smislu memorije. Nakon ove faze se prelazi u fazu *pregledanja traga*.

4.2 Pregledanje traga

Kada je naš program završio, moguće je pregledati *trag* korišćenjem alata koje nudi Het. Važno je napomenuti da se pod terminom "završavanje programa" ne smatra da je program isključivo završio *ispravno*, već da je program eventualno završio sa nekom porukom o grešci ili pak da je prekinut od strane programera [9]. Za analizu *traga* Het nudi nekolicinu interaktivnih alata koji pregledaju ponašanje programa, između ostalog su to: *hat-observe*, *hat-trail*, *hat-detect*, *hat-explore* i *hat-stack*. Opis navedenih alata se nalazi u tabeli 1. Veliki broj alata koji nude različite poglede na program koji se debaguje je jedna od glavnih osobina alata Het, a uzimajući u obzir da su određeni alati inspirisani nekim već postojećim debagerima (Freja (eng. *Freja*) kao inspiracija za *hat-detect* i Hud (eng. *HOOD*) kao inspiracija za *hat-observe* [9]), ređe se javlja potreba za drugim alatima pri debagovanju Haskel programa.

Nažalost, nekompatibilnost alata Het sa novijim verzijama Haskel biblioteka i *GHC*-om (*GHC - Glasgow Haskell Compiler*) ga čini nepri-

stupačnim programerima koji žele da posvete što manje vremena na iscrpna podešavanja verzija raznoraznih biblioteka, a što više na debagovanje.

Tabela 1: Opis nekih od alata koje nudi Het

Naziv alata	Opis
hat-observe	Prikazuje kako se koriste funkcije najvišeg nivoa, tj. za svako ime funkcije prikazuje sve argumente sa kojima je data funkcija pozivana u toku izračunavanja programa, kao i rezultate tih poziva [9].
hat-trail	Omogućava praćenje izračunavanja *unatraške*, počevši od poruke o grešci ili od izlaza programa [9].
hat-detect	Postavljanjem da/ne pitanja za svaku primenu vrednosti na neku funkciju, ovaj alat poluautomatski locira grešku u programu [9]. Debagovanje na ovaj način predstavlja srž *algoritamskog debagovanja*.
hat-stack	Ovaj alat za neuspešna izvršavanja programa nagoveštava u kojoj funkciji je došlo do prekida izvršavanja, i to tako što ispiše *virtuelni stek*[1] funkcijskih poziva [9].
hat-explore	Slično kao i kod uobičajenih debagera, ovaj alat označava trenutnu poziciju u izvornom kodu u kojoj se nalazi prilikom izračunavanja programa, ujedno prikazujući i redosled pozivanja funkcija u toku izračunavanja [9].

5 HOOD Debager

Hud (eng. *HOOD – Haskell Object Observation Debugger*) je mali debager za Haskel, baziran na posmatranju struktura podataka dok se prosleđuju između funkcija.[2] Implementiran je kao nezavisna biblioteka koju je moguće koristiti iz bilo kog Haskel kompajlera, što ga izdvaja u odnosu na ostale debagere. Korišćenje huda je relativno prosto. Prvo se vrši umetanje funkcije *observe* ispred objekta koji se posmatra ili između dve funkcije čije međustanje želimo da posmatramo. Nakon toga program se izvršava, a zatim se vrši ispis stanja objekata koji su posmatrani. Tip ove funkcije je:

```
1000 observe :: (Observable a) => String -> a -> a
```

gde je prvi argument labela kojom obeležavamo ispis, a drugi je objekat koji se posmatra. Kao što se vidi u potpisu funkcije, postoji tipsko ograničenje tj. posmatrani objekat mora da bude klase *Observable*, što je već implementirano za osnovne tipove. Za svaki novi tip koji se napravi nepohodno je da pridruži ovoj klasi, ako želimo da ga posmatramo. Što se

[1] Stek je *virtuelni* zato što je u stvarnom steku izračunavanja Haskel programa omogućeno *lenjo izračunavanje*, dok se kod virtuelnog steka prikazuje kakav bi bio stek u slučaju strogog izračunavanja.

tiče Haskela, *observe* se ponaša kao funkcija identiteta s tim što čuva podatke za kasnije čitanje. U jednom programu je moguće imati više poziva funkcije observe, koje razlikujemo korišćenjem labela. Takođe je moguće posmatrati bilo koji izraz, a ne samo međustanja funkcijskih poziva. Pozivi funkcije *observe* ne zahtevaju dodatna izračunavanja posmatranog objekta što ide u prilog efikasnosti ovog alata.[6] Glavna prednost huda je to što uz minimalne promene kôda dobijamo strukturiran prikaz objekata.

5.1 Primeri korišćenja funkcije observe

Primer 5.1 *Posmatranje konačne liste:*

Ovde je eksplicitno naveden tip podatka koji se posmatra, ali to nije neophodno.

```
1000 pr1 :: IO ()
     pr1 = print ((observe "lista" :: Observing [Int])[0..9])
1002 -- lista
     0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : []
```

Podjednako je validan i sledeći izraz:

```
1000 pr1 = print (observe "lista" [0..9])
```

Primer 5.2 *Posmatranje međustanja*

```
1000 pr2 :: IO()
     pr2 = print . reverse . observe "medjustanje" . reverse $ [0..9]
1002 -- medjustanje
     9 : 8 : 7 : 6 : 5 : 4 : 3 : 2 : 1 : []
```

Ovde vidimo da observe podržava parcijalnu aplikaciju, što je standardni način zapisivanja kada posmatramo međustanja.[2]

Primer 5.3 *Beskonačne liste*

```
1000 pr3 :: IO()
     pr3 = print (take 6 (observe "beskonacna lista" [0..]))
1002 -- beskonacna lista
     0 : 1 : 2 : 3 : 4 : 5 : _
```

Primećujemo da su brojevi od 0 do 5 izračunati i prikazani, a ostatak koji nije izračunat je prikazan sa karakterom _.

Primer 5.4 *Liste sa neizračunatim elementima*

```
1000 pr4 :: IO()
     pr4 = print (length (observe "konacna lista" [1..6]))
1002 -- konacna lista
     _ : _ : _ : _ : _ : _ : []
```

Pošto je Haskell lenj jezik, elementi nisu izračunati, pa samim tim ni observe ne može da ih vidi.

Primer 5.5 *Liste sa polovično izračunatim elementima*

```
pr5 :: IO ()
pr5 = let xs = observe "polovicna lista" [0..9]
  in print(xs !! 1 + xs !! 5)
-- polovicna lista
_ : 1 : _ : _ : _ : 5 : _
```

Primećujemo da observe vidi samo one elemente koji su izračunati.

Primer 5.6 *Korišćenje više funkcija observe*

U ovom primeru vidimo kako možemo da ispratimo sva međustanja izvršavanja jedne funkcije, što znatno olakšava uočavanje mesta greške. Funkcija vraća niz cifara datog broja.

```
cifre :: Int -> [Int]
cifre = observe "posle reverse"
    . reverse
    . observe "posle map"
    . map (`mod` 10)
    . observe "posle takeWhile"
    . takeWhile (/= 0)
    . observe "posle iterate"
    . iterate (`div` 10)
cifre 3542
-- posle iterate
(3542 : 354 : 35 : 3 : 0 : _)
-- posle takeWhile
(3542 : 354 : 35 : 3 : [])
-- posle map
(2 : 4 : 5 : 3 : [])
-- posle reverse
(3 : 5 : 4 : 2 : [])
```

Primer 5.7 *Posmatranje funkcija*

Pored posmatranja osnovnih tipova podataka, moguće je i posmatranje funkcija tj. posmatranje mapiranja argumenata u rezultate.[2] Argumenti i rezultati mogu sadržati neizračunate elemente, kao u prethodnim primerima.

```
pr6 = print ((observe "length" :: Observing([Int] -> Int))
  length [1..3])
-- length
{ \ (_ : _ : _ : [] -> 3
}
```

Funkcija *observe* sada prima tri argumenta: labelu, funkciju(length) i njen argument. Ovako Haskel program tumači ovaj izraz:

```
(observe "length" :: Observing ([Int] -> Int)) length [1..3]
-- uklanja se anotacija tipa posmatrane funkcije
observe "length" length [1..3]
-- observe i labela "length" se zamenjuju funkcijom identiteta
id length [1..3]
-- id uzima jedan  argument
(id length) [1..3]
-- id length postaje samo length
length [1..3]
```

Ovakvo tumačenje podržava i funkcije sa više argumenata kao i funkcije višeg reda.

```
pr7 = print (observe "foldl (+) 0 [1..4]" foldl (+) 0 [1..4])
-- foldl (+) 0 [1..4]
{ \ { \ 6 4 -> 10
    , \ 3 3 -> 6
    , \ 1 2 -> 3
    , \ 0 1 -> 1
    }
    0
    (1 : 2 : 3 : 4 : [])
    -> 10
}
```

Posmatrajući *foldl*, posmatrali smo i njene argumente i dobili detaljan prikaz izvršavanja.

Sada ćemo razmotriti prethodni primer sa ciframa, samo što ćemo ovog puta posmatrati funkcije umesto međustanja.

```
cifre :: Int -> [Int]
cifre = reverse
  . observe "map" map (`mod` 10)
  . observe "takeWhile" takeWhile(/= 0)
  . observe "iterate" iterate (`div` 10)
-- iterate
{  \  {  \ 3 -> 0
       ,  \ 35 -> 3
       ,  \ 354 -> 35
       ,  \ 3542 -> 354
       } 3542
       -> 3542 : 354 : 35 : 3 : 0 : _
}
-- takeWhile
{  \  { \ 0 -> False
       , \ 3 -> True
       , \ 35 -> True
       , \ 354 -> True
       , \ 3542 -> True
       } (3542 : 354 : 35 : 3 : 0 : _)
       -> 3542 : 354 : 35 : 3 : []
}
-- map
{  \  {  \ 3 -> 3
       ,  \ 35 -> 5
       ,  \ 354 -> 4
       ,  \ 3542 -> 2
       } (3542 : 354 : 35 : 3 : [])
       -> 2 : 4 : 5 : 3
```

Funkcija *iterate* je uzela broj 3542 i napravila beskonačni opadajući niz brojeva od kojih je samo prvih pet izračunato. Funkcija *takeWhile* je od beskonačnog niza napravila konačni kada je naišla na element 0. Funkcija *map* je od svakog elementa niza uzela poslednju cifru i napravila novi niz koji funkcija *reverse* obrće. U ovom primeru vidimo koliko je hud moćan alat i sa kojom se lakoćom koristi.

6 Debagovanje korišćenjem Debug biblioteke

Debug biblioteka je kreirana or strane Nila Mičela(eng. *Neil Mitchell*) zarad laganog debagovanja Haskel programa[4]. Fokus ove biblioteke jeste

na jednostavnosti korišćenja i intuitivnom interfejsu. Pošto je u pitanju Haskel biblioteka, ona ne zavisi od eksternih alata, što znatno olakšava njeno korišćenje i održavanje. Debug pri korišćenju generiše trag (eng. *trace*) i omogućava jasno praćenje generisanog traga kroz svako pozivanje funkcije[1]. Mičelov Debug se može integrisati u Haskel program na više načina, najčešće putem enkapsuliranja programskog koda u okviru debug funkcije[4].

Program koji ćemo koristiti za demonstraciju Debug biblioteke se sastoji od 3 funkcije, sve tri treba da vrate koliko elemenata u nizu je veće od početnog.

```haskell
{-# LANGUAGE TemplateHaskell, ViewPatterns, PartialTypeSignatures
    #-}
{-# OPTIONS_GHC -Wno-partial-type-signatures #-}
import Debug

debug [d|
    countGreater1st :: (Ord a) => [a] -> Int
    countGreater1st [] = 0
    countGreater1st [x] = 0
    countGreater1st (x:y:xs)
            | y > x = 1 + countGreater1st (x:xs)
            | otherwise = 0 + countGreater1st (x:xs)

    countGreater1st' :: (Ord a) => [a] -> Int
    countGreater1st' [] = 0
    countGreater1st' [x] = 0
    countGreater1st' (x:y:xs)
            | y > x = 1 + countGreater1st' (y:xs)
            | otherwise = 0 + countGreater1st' (x:xs)

    countGreater1st'' :: (Ord a) => [a] -> Int
    countGreater1st'' [] = 0
    countGreater1st'' [x] = 0
    countGreater1st'' (x:y:xs)
            | y >= x = 1 + countGreater1st'' (x:xs)
            | otherwise = 0 + countGreater1st'' (x:xs)
    |]
```

Listing 1: Okružujemo naš kod funkcijom debug, iz biblioteke Debug, sa uključivanjem ekstenzija navedenih u prvom redu

6.1 Primena debug-a

U ovom odeljku će biti demonstriran tipičan primer korišćenja debug biblioteke, korišćenjem primera navedenog gore.

```
$ ghci example-Mitchell-debug.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( example-Mitchell-debug.hs, interpreted )
Ok, modules loaded: Main.
*Main> countGreater1st [2,3,4,5,6]
4
*Main> debugView
*Main>
```

Slika 2: Primer pozivanja ghci i debug putem terminala

Nakon što pozovemo bilo koju funkciju i ona se izvrši, ona zahvaljujući debug funkciji ostavlja trag. Komandom debugView pozivamo brauzer(browser) prozor koji nam prikazuje željene informacije. Druga opcija je da sa debugRun automatski izvršimo funkciju i pozovemo prozor[4].

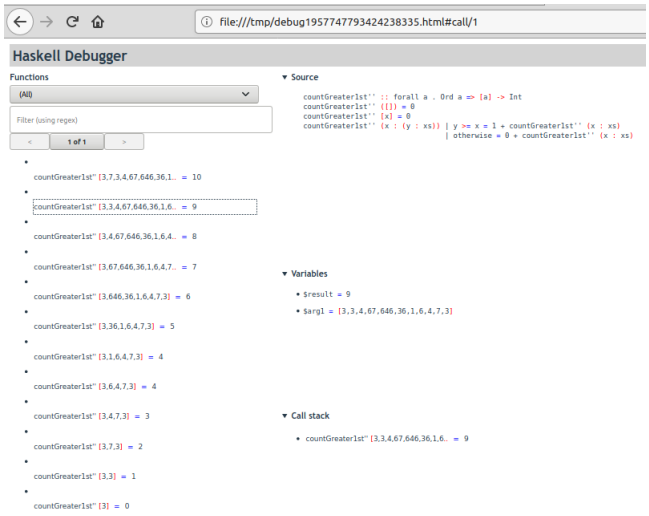
Slika 3: Prozor Debug-a nakon primene countGreater1st

Za svaku od levo navedenih funkcija koje predstavljaju call stack ovog programa možemo jasno videti argumente i rezultat, što nam omogućava pregledno debagovanje bilo kog Haskell programa. Primetićemo da prvi primer radi kako treba, tj vraća 8 elemenata većih od prvog.

Slika 4: Prozor Debug-a nakon primene countGreater1st'

Drugi primer vraća rezultat 3. Na nama je da vidimo u čemu je problem. U levoj strani možemo videti sve funkcije koje su pozvane, što nam služi kao Call Stack programa. Prvi red predstavlja prvu pozvanu funkciju. Drugi red nam je sledeći poziv. Primetimo da je neobičan. Naš plan s ovom funkcijom je da nađemo broj elemenata veći od prvog, a izgleda da smo ga odbacili pri drugom pozivu. Gore imamo programski kod funkcije i možemo pogledati šta je urađeno. Kad je drugi element veći od prvog, mi rekurzivno pozivamo funkciju sa listom bez prvog elementa, i tako kroz

ostale. Našli smo bug!



Slika 5: Prozor Debug-a nakon primene countGreater1st", izabrali smo drugi poziv da pogledamo

U trećem primeru dobijamo krajnji rezultat 10. Primetimo u drugom pozivu, gledajući call stack, da on vraća +1 za niz koji počinje sa 3,3. Gledajući kod funkcije primetimo da vraća +1 kada su jednaki brojevi. Bug je pronađen!

Ovo je u suštini kako se radi sa Debug-om. Jednostavan prozor gde se vidi manje-više sve što treba.

6.2 Problemi debug-a i Debug.Hoed

Debug nije bez svojih problema. On koristi Show instance da bi prikazao vrednosti, što pravi probleme ako se program oslanja na lenjo izračunavanje, na primer kad imamo beskonačni niz. U tom slučaju program će najčešće da crash-uje ili se zaglavi u beskonačnoj petlji[4].

Debug.Hoed rešava gorenavedene probleme. U pitanju je biblioteka građena na Debug koja takođe koristi TemplateHaskell, koja podržava lenjo izračunavanje i nudi jasan prikaz Call Stack-a. Primer kako izgleda prozor s njim se može pronaći u literaturi. Važno je napomenuti da je u eksperimentalnoj fazi, tj. mogu se očekivati drugi bagovi pri korišćenju.

Nažalost, nismo u mogućnosti da prikažemo praktičan primer sa Debug.Hoed usled problema sa zastarelom verzijom ghci koja se dobija preko apt repozitorijuma(8.0 umesto 8.2+, koji Debug.Hoed zahteva)

7 Zaključak

Ovaj rad se pozabavio pitanjem debagovanja u Haskel programskom jeziku. Posle analize i isprobavanja svih debagera, stekli smo bolje razumevanje u to kako debagovanje u Haskelu funkcioniše. No glavno pitanje koje je potrebno naglasiti kada je u pitanju debagovanje u Haskelu je – da li je debagovanje u Haskelu prijemčivo? Autori rada smatraju da je

sam Haskelov dizajn kao programskog jezika zajedno sa svojim ugrađenim debagerima sasvim dovoljan kada je u pitanju debagovanje, što se može potvrditi čestom pojavom da debageri izvan već ugrađenog nisu ažurni sa trenutnom verzijom jezika, ili prosto uopšte ne funkcionišu više. Među ostalom kao primer se može navesti većina izloženih tehnologija u ovom radu, koje su u velikom broju slučajeva ili nefunkcionalne ili zapostavljene.

Literatura

[1] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming haskell for tracing. In *Symposium on Implementation and Application of Functional Languages*, pages 165–181. Springer, 2002.

[2] Andy Gill. Debugging haskell by observing intermediate data structures.

[3] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[4] Neil D. Mitchell. Mitchell Debug, 2017. on-line at: https://github.com/ndmitchell/debug.

[5] Bernard James Pope. *A declarative debugger for Haskell*. PhD thesis, 2006.

[6] Bernard James Pope. A declarative debugger for haskell. 2006.

[7] Bernie Pope. Step inside the ghci debugger. *Monad Reader*, 1(10), 2008.

[8] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:1–5, 2002.

[9] Hat Team. The Haskell Tracer Hat, 2013. on-line at: https://archives.haskell.org/projects.haskell.org/hat/.

[10] Simon Thompson. *The Craft of Functional Programming*. Addison-Wesley, London, 1999.

[11] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for haskell: a new hat. 2001.