UNIVERZA V LJUBLJANI

FAKULTETA ZA MATEMATIKO IN FIZIKO

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Ivo List

Plavajoča vejica v čistem funkcijskem jeziku

DIPLOMSKO DELO
NA INTERDISCIPLINARNEM UNIVERZITETNEM ŠTUDIJU

MENTOR: prof. dr. Andrej Bauer

Ljubljana, 2016

Copyright © 2016 Ivo List.
Ta dokument lahko razmnožujete, razširjate in/ali spreminjate pod pogoji Dovoljenja za prosto dokumentacijo GNU (GNU Free Documentation License), različice 1.3 ali katerekoli poznejše različice, ki jo je izdala ustanova Free Software Foundation; brez Nespremenljivih razdelkov, brez Besedila na naslovnici in brez Besedila na zadnji strani.
Izvod licence je dosegljiv na: `http://www.gnu.org/licenses/fdl-1.3.en.html`.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is available on: `http://www.gnu.org/licenses/fdl-1.3.en.html`.

Tema diplomskega dela

„V delu predstavite plavajočo vejico s poljubno fiksno natančnostjo. Knjižnico Multiple Precision Floating-Point Reliable Library (MPFR) povežite s programskim jezikom Haskell in implementirajte razrede za delo s plavajočo vejico s poljubno natančnostjo. Primerjajte učinkovitost vaše implementacije s standardno uporabo knjižnice MPFR v C."

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Ivo List,

z vpisno številko 63040484,

sem avtor/-ica diplomskega dela z naslovom:

Plavajoča vejica v čistem funkcijskem jezik

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  prof. dr. Andreja Bauerja

  in somentorstvom (naziv, ime in priimek)

  /

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 7.2.2016 Podpis avtorja/-ice: ____________________

Kazalo

Povzetek **9**

1 Uvod **11**

2 Plavajoča vejica **13**

2.1 Predstavitev 15
2.2 Normalna oblika 15
2.3 Subnormalna oblika 16
2.4 Strojna oblika – IEEE 754 16
2.5 Posebne vrednosti 17
2.6 Zaokroževanje 18
2.7 Natančnost in točnost 19
2.8 Dilema izdelovalca tabel 20
2.9 Izjeme 20
2.10 Povratne pretvorbe 21
2.11 Knjižnice 23

3 Čisti funkcijski jeziki **25**

3.1 Neučakani in leni jeziki 25
3.2 Implementacija – GHC 26
3.2.1 Primitivni tipi in operacije 26
3.2.2 Uporaba kode iz drugih jezikov 28
3.2.3 C-- in Cmm 28
3.2.4 Profiliranje 29
3.2.5 Upravljanje s pomnilnikom in odstranjevanje smeti 29
3.2.6 Predstavitev literalov 30

4 Uporaba knjižnice MPFR v GHC-ju **31**

4.1 MPFR 31
4.1.1 Zaokroževanje 32

4.1.2 Rezultat na posebnih številih 32
4.1.3 Izjeme 33
4.1.4 Upravljanje s pomnilnikom 33
4.1.5 Nomenklatura in tipi 34
4.1.6 Dogovori o spremenljivkah 34
4.1.7 Funkcije knjižnice 34
4.2 Funkcijski vmesnik 35
4.3 Predstavitev MPFR vrednosti v GHC 37
4.4 Prenos funkcij z enako signaturo 38
4.5 Funkcije za pretvorbo in literali 39
4.6 Uporaba Cmm in neuokvirjenih tipov 39
4.7 Primerjava hitrosti 40

5 Zaključek **43**

Povzetek

Naslov: Plavajoča vejica v čistem funkcijskem jeziku

Opisana je plavajoča vejica, njena predstavitev, standarda IEEE 754, posebne vrednosti, načini zaokroževanja in nekateri problemi s katerimi se soočimo pri njeni implementaciji. Mednje sodita dilema izdelovalca tabel in problem povratnih pretvorb. Knjižnica Multiple Precision Floating-Point Reliable Library (MPFR) je implementacija plavajoče vejice s poljubno fiksno natančnostjo, ki jo želimo uporabiti v čistem funkcijskem jeziku. Predstavnik slednjega je Haskell in sledijo podrobnosti ene izmed njegovih implementacij – GHC. Zaradi učinkovitosti je potrebno dobro razumevanje upravljanja s pomnilnikom in najnižjih stopenj prevajalnika. Z navedenim je izdelan vmesnik za MPFR v GHC-ju in predstavljena je njegova primerjava s C-jevo implementacijo.

Abstract

Title: Floating-point in a purely functional language

The floating-point, its presentation, IEEE 754 standards, special values, rounding modes, and some problems with its implementation are presented: The Table Maker's Dilemma and Round-trip conversions. Arbitrary fixed precision floating-point is an extension of floating-point implemented in Multiple Precision Floating-Point Reliable Library (MPFR). Working towards an interface for a purely functional language, Haskell as a representative is introduced. For an efficient implementation good knowledge of one of its implementation – GHC – is needed, especially how memory management is done and how low-level stages of the compiler are implemented. Last, the interface for the GHC and its benchmark with C implementation is presented.

Ključne besede: plavajoča vejica, funkcijsko programiranje, Haskell, GHC, MPFR, C

Keywords: floating-point, functional programming, Haskell, GHC, MPFR, C

1 Uvod

Danes ima strojno implementacijo števil v plavajoči vejici vsak osebni računalnik in tudi marsikatera grafična kartica. Uporaba je za programerja enostavna in dosegljiva v najbolj pogostih programskih jezikih. Kljub temu pa se uporabniki redko zavedajo podrobnosti, na katere morajo paziti, in omejitev, ki jih ima strojna implementacija. Nekatere izmed teh omejitev lahko zaobidemo z uporabo plavajoče vejice s poljubno fiksno natančnostjo.

Slednja nima močno razširjene strojne implementacije, je pa implementirana v nekaterih programskih knjižnicah. Pri izbiri knjižnice nas vodi predvsem učinkovitost in izberemo Multiple Precision Floating-Point Reliable Library (MPFR). Le-ta je napisana v C-ju in želeli bi jo uporabiti tudi v drugih jezikih. Posebno težavo pri tem predstavljajo čisti funkcijski jeziki, kot je Haskell. Za prenos knjižnice potrebujemo dobro razumevanje Haskell-a, predvsem njegove implementacije.

V naslednjem poglavju predstavimo, kaj je plavajoča vejica, kako jo lahko razširimo ter najpogostejše težave na katere moramo biti pozorni. V tretjem poglavju so predstavljeni funkcijski jeziki, Haskell in nekatere podrobnosti njegove implementacije. V zadnjem poglavju predstavimo knjižnico MPFR in opišemo, kako smo naredili učinkovit vmesnik zanjo v Haskell-u.

2 Plavajoča vejica

Ker so računalniki omejeni s pomnilnikom in časom, lahko števila zapišemo le s končnim številom simbolov. Običajno cela števila omejimo na določeno število bitov oziroma števk v dvojiškem sistemu, ki ustrezajo dolžini registrov. Taka števila na računalnikih delujejo najhitreje, vendar to pripelje do zahtevnejših programerskih tehnik, saj je ob morebitni *prekoračitvi obsega* (*angl. overflow*) potrebno pravilno obravnavati izjeme.

Pravzaprav se redko zgodi, da bi bile te izjeme obravnavane in to lahko povzroči nepričakovano nepravilno delovanje programov. Pred nevarnostmi nas lahko reši razmislek ali celo dokaz, da do prekoračitve ne bo prišlo.

Da se izognemo napakam ali kadar potrebujemo večja števila, se ne omejimo na določeno število bitov. Zapis je običajno sestavljen iz dolžine in iz niza besed, ki predstavljajo število. Tako dobimo *cela števila s poljubno natančnostjo* (*angl. arbitrary-precision integer*). V tem primeru se hitrost operacij občutno zmanjša, saj so le-te le delno strojno podprte in tudi celotnega števila ne moremo več shraniti v en register. Prednost je, da nam ni potrebno posebno obravnavanje izjem zaradi prekoračitev.

Poleg naravnih števil bi seveda želeli računati tudi z racionalnimi števili oziroma predstaviti približke realnih števil. *Racionalna števila* lahko zapišemo kot par celih števil – ulomek. Če pri tem uporabimo števila s poljubno natančnostjo, je problem na videz rešen, saj lahko točno izračunamo vse aritmetične operacije.

Sicer v splošnem rezultata funkcije, definirane na realnih številih, ne moremo izraziti z racionalnimi števili. Namesto točnega rezultata zapišemo racionalno število v njegovi okolici. Temu rečemo *zaokroževanje*.

Aritmetične operacije z racionalnimi števili niso najhitrejše, saj na primer za eno množenje potrebujemo dve množenji celih števil in po potrebi krajšanje. Prav tako lahko že pri osnovnih aritmetičnih operacijah velikosti števca in imenovalca hitro zrasteta. Problematična je tudi primerjava dveh racionalnih števil, saj so zanjo potrebni dve množenji in odštevanje.

Rešitev teh problemov je, da ne zapišemo vseh racionalnih števil, temveč samo nekatera. V preteklosti so se pogosto uporabljala tako imenovana *števila s fiksno vejico* (*angl. fixed point*). Pri teh je imenovalec postavljen na v naprej določeno vrednost, ki se vodi

implicitno (ni zapisana). Pri računanju z valutami lahko na primer uporabimo imenovalec 100. Tak zapis močno pohitri delovanje operacij. Danes se zato uporablja na vgrajenih sistemih (*angl. embedded systems*) ali v aplikacijah, kjer je uporaba določena s standardi oziroma z zakonom. Hitrost ima tudi svojo ceno, saj moramo tudi aritmetične operacije zaokrožiti.

Eden od problemov števil v fiksni vejici je v tem, da števila niso gosta. Vrednosti med dvema številoma, ki imata števec za 1 različen, ne moremo predstaviti. Rešitev je v znanstvenem zapisu števila, npr. $3{,}12 \cdot 10^{-5}$. Zopet števila zapišemo kot pare celih števil, le da imata števili drug pomen kot pri racionalnih številih.

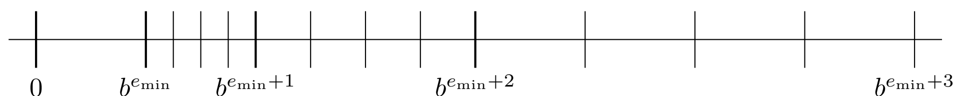
Števila zapišemo v obliki $m \cdot b^e$, kjer b imenujemo *osnova eksponenta* (*angl. base*) in je vnaprej dogovorjena ter se vodi implicitno. Najpogosteje je osnova 2, 10 ali 16. Številu m rečemo mantisa ali signifikand in številu e eksponent. Interno je tako število predstavljeno kot par celih števil (m,e).

Oblika števil, kjer sta m in e omejena na določeno število bitov imenujemo *zapis v plavajoči vejici* (*angl. floating point*) (slika 2.1). Le-ta je danes najpogostejši in operacije z njim imajo dobro strojno podporo in so zato zelo hitre. Ime izvira iz dejstva, da so vse neničelne števke števila zapisane v mantisi, medtem ko z eksponentom določamo položaj vejice med njimi in tako vejica plava.

Pri naravnih številih smo omenili težave s prekoračitvijo obsega. V plavajoči vejici do tega pride, ko je eksponent rezultata večji od največje predstavljive vrednosti z danim številom bitov. Izjemo do katere pride, kadar je eksponent rezultata premajhen za našo predstavitev, imenujemo *podliv* (*angl. underflow*).

Vrednosti med dvema številoma sedaj lahko izrazimo, dokler nam to dovoljuje predstavitev mantise in v skrajnih primerih eksponenta. S tem da mantise ne omejimo na določeno število bitov, dobimo predstavitev v *plavajoči vejici s poljubno natančnostjo* (*angl. arbitrary-precision floating point*). *Natančnost* je število bitov mantise. S tem (neomejeno mantiso in omejenim eksponentom) smo se zadovoljivo približali potrebi, da je naša predstavitev gosta.

V matematiki za množico števil v plavajoči vejici ni splošno dogovorjene oznake. V nadaljevanju bomo uporabljali oznako $\mathbb{D}$, kot asociacijo na angleško besedo double ali dyadic. Za števila v plavajoči vejici se pogosto, predvsem v računalniški literaturi, uporablja izraz realna števila. To seveda ni pravilno, saj so lahko števila v plavajoči vejici zgolj približki realnih števil. So pa števila v plavajoči vejici podmnožica realnih števil $\mathbb{D} \subset \mathbb{R}$. Razlikovanje je pomembno tudi zaradi sledečih definicij.



Slika 2.1: Števila v plavajoči vejici v okolici ničle z natančnostjo 3 bitov. Z b je označena osnova eksponenta, e_{min} je najmanjši predstavljiv eksponent. Povzeto po [?].

2.1 Predstavitev

Predstavitev števil (*angl. representation*) je način, kako števila zapišemo s simboli. Tu se bomo omejili na dvojiške simbole (0 in 1), saj je taka predstavitev uporabna za strojno implementacijo.

Naravna števila predstavimo z zapisom v dvojiški osnovi. Pri predstavitvah celih številih (z omejenim številom bitov) imamo na voljo več možnosti:

- 1 bit za predznak in absolutna vrednost števila,
- eniški komplement,
- dvojiški komplement in
- predstavitev z zamikom (celemu številu prištejemo vnaprej dogovorjeno vrednost) (*angl. offset/bias*).

Predstavitve celih števil s poljubno natančnostjo najpogosteje predznak zapisujejo kot poseben bit. To poenostavi algoritme za množenje in deljenje.

Pri predstavitvah števil v plavajoči vejici je pogosto uporabljen en način za mantiso in drugi za eksponent. V nadaljevanju si bomo ogledali različne možnosti za predstavitev le-teh.

2.2 Normalna oblika

Mantisa in eksponent nista enolično določena. Na primer, če mantiso pomnožimo z osnovo eksponenta in odštejemo 1 od eksponenta, predstavljata nova mantisa in eksponent isto število: $m \cdot b^e = mb \cdot b^{e-1}$. Da je predstavitev enolična, potrebujemo dodaten pogoj.

Ena izmed možnosti za ta pogoj je $1 \leq |m| < b$. Kljub temu, da je m zapisan kot celo število, ga običajno interpretiramo kot racionalno, kjer je vejica postavljana za prvim bitom oziroma na začetku kadar je le-ta zapisan implicitno. S tako interpretacijo je vrednost eksponenta neodvisna od natančnosti. Posebnost je seveda število 0. Števila, ki ustrezajo pogoju, imenujemo *normalizirana*. Dodatna prednost tega pogoja v dvojiškem zapisu je, da je vodilna števka vedno 1 in lahko tako prihranimo en bit.

2.3 Subnormalna oblika

Iz slike 2.1 vidimo, da je v okolici ničle razmik nesorazmerno velik. Kadar delamo s poljubno natančnostjo in velikim območjem eksponenta, to ni težava. Vendar je pri fiksni natančnosti bolje, da tudi ta prostor izkoristimo.

Razmik v okolice ničle se zgodi pri najmanjšem eksponentu. Povzroči ga zapis v normalni obliki. Odpravimo ga tako, da definiramo posebno, tako imenovano *subnormalno obliko* (*angl. subnormal form* oz. staro ime iz leta 1985 *denormalized form*).

V tej posebni obliki števila niso normalizirana, temveč imajo fiksen eksponent. Absolutna vrednost mantise je manjša od 1 in zato je vodilna števka 0, ki jo zapišemo implicitno. Posebno obliko števila zakodiramo kar z do sedaj najmanjšo vrednostjo eksponenta (v predstavitvi z odmikom je to 0). Fiksen eksponent, ki ga uporabljamo pri interpretaciji števila je za ena večji. Vrzel okoli ničle tako enakomerno zapolnimo.

2.4 Strojna oblika – IEEE 754

IEEE standard za dvojiško aritmetiko v plavajoči vejici (*angl. IEEE Standard for Binary Floating-Point Arithmetic*), kratko IEEE 754, je nastal leta 1985. Avgusta 2008 ga je nadomestil prenovljen standard. Standard definira zapis števil kot tudi posebne vrednosti, izjeme, načine zaokroževanja in operacije.

IEEE 754 standard definira 16, 32, 64 in 128[1] bitne dvojiške osnovne oblike zapisa ter dodatne razširjene oblike. Osnovne oblike so namenjene izmenjavi števil, medtem ko je razširjena oblika rezervirana za interno uporabo pri izračunu operacij.

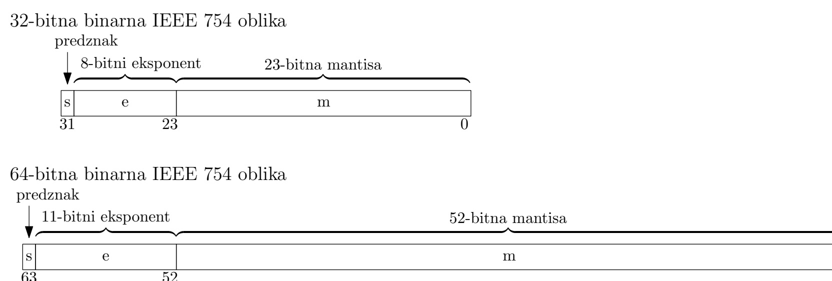
Pri vseh oblikah je mantisa zapisana z bitom za predznak in absolutno vrednostjo ter eksponent z zamikom. Pregled oblik je v tabeli 2.1. Poleg tega je natančno definirana tudi lega posameznih bitov v pomnilniku ali pri izmenjavi (prikazana na sliki 2.2). Ker je ta oblika implementirana tudi v strojni opremi, je računanje z njo zelo hitro.

Oblika	Število bitov			Odmik eksponenta
	skupaj	mantisa	eksponent	
enojna natančnost	32	23	8	127
dvojna natančnost	64	52	11	1023
razširjena	80	64	15	16383

Tabela 2.1: Pregled oblik zapisov v plavajoči vejici definiranih v standardih IEEE 754.

Standard zahteva implementacijo osnovnih aritmetičnih operacij, kvadratnega korena, ostanka in združenega množenja-seštevanja. Navedene operacije morajo biti tudi pravilno

[1] 16 in 128 bitni dvojiški obliki sta bili dodani leta 2008, kot tudi decimalne oblike



Slika 2.2: Razporeditev strojnih oblik števil v plavajoči vejici v pomnilniku ali registrih kot je definiran v standardih IEEE 754.

zaokrožene. Priporočena je implementacija še približno 50 drugih matematičnih funkcij. Poleg tega so zahtevane tudi primerjave, pretvorbe, testi in podobno.

2.5 Posebne vrednosti

Poleg številskih vrednosti lahko zapišemo tudi posebne vrednosti. Te so:

- pozitivna in negativna ničla,
- pozitivna in negativna neskončnost,
- več neštevilskih vrednosti, imenovanih NaN, ki je okrajšava za „ne število” (*angl. not a number*).

Ničla je zapisana z ničelno mantiso in najmanjšim eksponentom, tako da jo lahko interpretiramo tudi kot subnormalno število. Posebnost je zato, ker imamo pozitivno in negativno ničlo. Le-ti sta v primerjavah enaki in nista različni. Aritmetične operacije pri ničelnem rezultatu izberejo njegov predznak glede na to ali se rezultat v limiti približuje ničli od zgoraj oziroma od spodaj.

Pozitivna in negativna neskončnost sta dodani z razlogom, da se izjeme ne prožijo prepogosto in da ni potrebno ročno obravnavati vseh posebnih primerov. Ustrezna neskončnost se na primer vrne pri deljenju neničelnega števila z nič. Kadar neskončnosti nastopata kot argument pa se v primerih, ko limita obstaja vrne le-ta.

Kadar rezultat operacije ni definiran niti v limiti se vrne posebna vrednost NaN, npr. pri deljenju nič z nič. NaN nima predznaka. Posebno vlogo NaN igra v primerjavah. Vse primerjave z NaNom vrnejo „false”, razen posebne operacije (*angl. „unordered”*), ki vrne „true” v primerih, ko je en operand NaN.

NaN, ki se vrne v operacijah, je tako imenovani tihi NaN (*angl. quiet NaN*) oziroma qNaN. Poleg tega obstaja še signalni sNaN, ki ga nastavimo sami in lahko vsebuje dodatno informacijo. Ko se pojavi v operacijah, signalni NaN vedno proži izjemo.

2.6 Zaokroževanje

Tudi v plavajoči vejici s poljubno natančnostjo želimo v posameznih stopnjah izračuna natančnost omejiti. Poleg tega rezultata nekaterih operacij nad realnimi števili ne moremo izraziti pri poljubno veliki natančnosti. Zato moramo definirati, kako rezultat operacij zapisati s približkom.

Preden se je pojavil standard IEEE 754/85 ni bilo natančne definicije, kako to storiti. Lahko si predstavljamo, kakšne posledice je to imelo. Natančna definicija nam pomaga, da vemo kakšen rezultat pričakovati in da lahko isto pričakujemo tudi na drugi strojni opremi.

Preslikavi $\circ : \mathbb{R} \to \mathbb{D}$ iz realnih števil v števila s plavajočo vejico rečemo *zaokroževanje*. Na primer točen rezultat seštevanja $a + b : \mathbb{R}$ zapišemo kot $\circ(a + b) : \mathbb{D}$.

Izkaže se, da lahko tako preslikavo definiramo na več uporabnih načinov. Izbiri preslikave pravimo *način* oziroma *pravilo zaokroževanja* (*angl. rounding mode/rule*).

Ker se taka preslikava uporablja pri vseh operacijah, je smiselno namesto parametra uporabiti globalni atribut. Globalni atribut je uporabljen v IEEE 754/85 standardu.

V standardu so definirani naslednji načini zaokroževanja (prikazani na sliki 2.3):

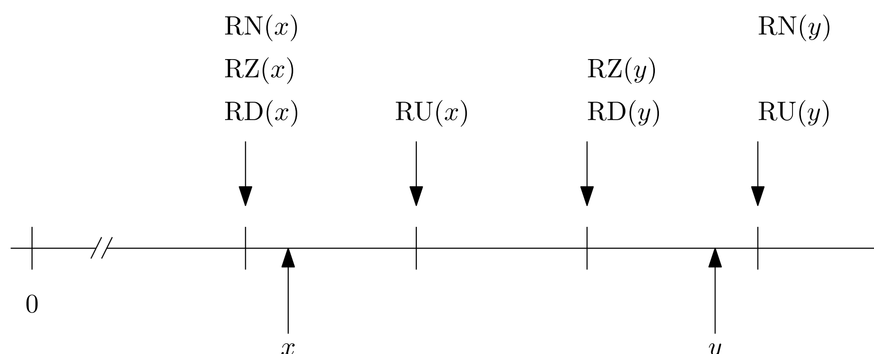
- *usmerjena* zaokroževanja:
  - zaokroževanje proti $-\infty$: $\mathrm{RD}(x)$ je največje število v plavajoči vejici, ki je manjše ali enako x,
  - zaokroževanje proti $+\infty$: $\mathrm{RU}(x)$ je najmanjše število v plavajoči vejici, ki je večje ali enako x,
  - zaokroževanje proti ničli: RZ(x) je število v plavajoči vejici, ki je najbližje x in absolutne vrednosti nima večje od x,
- *zaokroževanja k najbližjemu številu*: $\mathrm{RN}(x)$ je število v plavajoči vejici, ki je najbližje x. V primeru, da obstajata dve taki števili (obe z enako razdaljo do x), velja eno od naslednjih pravil:
  - zaokroževanje k najbližjemu, sodemu številu: v pat situaciji uporabimo najbližje sodo število. To je privzet način zaokroževanja v IEEE 754-2008,
  - zaokroževanje k najbližjemu, stran od ničle: v pat situaciji uporabimo število, ki je dlje od ničle.

Katera izbira načina zaokroževanja je najboljša, je odvisno od primera. Pri finančnih izračunih moramo običajno uporabiti poseben način predpisan z računovodskimi standardi, medtem ko lahko za numerične izračune, dobro argumentiramo izbiro zaokroževanja

k najbližjemu sodemu številu. Morda je videti, da se preveč poglabljamo v podrobnosti, vendar se izkaže, da se primeri, ko je rezultat ravno v sredini med dvema številoma, pojavljajo pogosteje, kot bi pričakovali.

Oglejmo si enostaven primer v desetiškem sistemu s tremi mesti in uporabimo pravilo, da se rezultat vedno zaokroži navzgor. Začnimo s številom $u = 1{,}00$, ki mu prištejemo in odštejemo število $v = 0{,}555$. Po seštevanju dobimo 1,56 in odštevanju 1,01. Ko operaciji ponovimo, dobimo števila 1,02, 1,03, in tako naprej. Pojavu pravimo *drsenje* (*angl. drift*) [?].

Do tega pojava ne pride, kadar uporabljen način zaokroževanja k najbližjemu rešuje pat situacij glede na sodost oziroma lihost [?].



Slika 2.3: Štirje načini zaokroževanja v primeru, da sta x in y pozitivni števili. Povzeto po [?].

Beseda zaokroževanje se včasih uporablja površno, zato pravimo, da kadar je eksakten rezultat operacije (izračunan z neskončno natančnostjo in na neomejeni sliki) zaokrožen glede na način zaokroževanja, tudi *pravilno* ali *eksaktno zaokrožen.*

2.7 Natančnost in točnost

Natančnost (*angl. precision*) je število števk mantise, kot smo že omenili v prejšnjih razdelkih.

Za točnost (*angl. accuracy*) je v uporabi več definicij. Mathematica za definicijo točnosti uporablja število decimalnih mest, t. j. število znanih števk desno od vejice, če število zapišemo z vejico brez eksponenta [?]. Slednje se uporablja, ko je rezultat blizu 0. V drugih virih se točnost uporablja kot absolutna napaka rezultata.

Absolutna napaka se pogosto izraža oziroma primerja z ULP-i (*angl. unit in the last place*). *ULP* je enota, katere vrednost je enaka najmanjši razliki med dvema različnima številoma v plavajoči vejici, ki imata enako natančnost in eksponent. ULP-i so odvisni tako od natančnosti kot od eksponenta. Pri usmerjenih načinih zaokroževanja je absolutna

napaka manjša od 1 ULP, medtem ko je pri načinih zaokroževanja k najbližjemu številu absolutna napaka manjša ali enaka $\frac{1}{2}$ ULP.

Izrazimo lahko tudi relativno napako. Relativna napaka zaokroževanja števil v plavajoči vejici s fiksno natančnostjo ima zgornjo mejo. Velikosti le-te rečemo tudi *strojni epsilon*.

2.8 Dilema izdelovalca tabel

Kako lahko funkcije sploh pravilno zaokrožimo? Za transcendentne funkcije (kot na primer: sin, cos, exp, log) ne moremo s števili v plavajoči vejici predstaviti eksaktnega rezultata. Rešitev je, da pred zaokroževanjem izračunamo funkcijo z večjo natančnostjo. To natančnost bomo imenovali *vmesna natančnost*, da jo lahko razlikujemo od natančnosti, ki jo želimo pri rezultatu. *Dilema izdelovalca tabel* (*angl. table maker's dilemma*) je določiti, kakšna je najmanjša vmesna natančnost, da bo rezultat pravilno zaokrožen. V kolikor vmesno natančnost povečamo, bo rezultat po zaokroževanju ostal enak.

Najmanjša vmesna natančnost je odvisna od argumentov in od končne natančnosti, ki jo želimo imeti. Najmanjšo natančnost pri vseh različnih parametrih za dano končno natančnost (npr. dvojno natančnost oz. 53 bitov) je težko določiti. Problema se lotimo tako, da poiščemo argumente, pri katerih je vmesna natančnost največja. Pri dvojni natančnosti je za logaritemsko funkcijo najslabši primer [?]:

$$\begin{aligned}\log 1\,1010\,1100{,}0101\,0000\,1011\,0100\,0000\,1001\,1100\,1000\,1010\,1110\,1110 = \\ 110{,}0000\,1111\,0101\,0010\,1111\,0011\,0111\,1010\,1110\,1100\,1111\,1100\,1111^{60}0101\ldots\end{aligned}$$

kjer z 1^n zaznamujemo n ponovitev števke 1. Tako za izračun logaritemske funkcije v dvojni natančnosti potrebujemo vmesno natančnost s 115 števkami.

2.9 Izjeme

Ko pride do izjeme je privzeto vedenje po IEEE 754 standardu, da se vrne ena od posebnih vrednosti in ne povzroči prekinitve. Obenem se nastavijo zastavice (*angl. flags*), ki jih uporabnik lahko prebere in ponastavi. Zastavice so lepljive, t. j. ostanejo nastavljene dokler jih uporabnik ne ponastavi. V določenih primerih lahko le s testiranjem zastavic ugotovimo ali je prišlo do izjeme.

V standardu so definirane naslednje izjeme:

- prekoračitev (*angl. overflow*),
- podliv (*angl. underflow*),

- deljenje z ničlo,
- napačna operacija (*angl. invalid operation*) in
- neeksakten rezultat (*angl. inexact*).

Prekoračitev se zgodi, kadar je rezultat večji od največjega še predstavljivega števila ali kadar je rezultat (negativen in) manjši od najmanjšega predstavljivega števila. Lahko tudi rečemo, da je eksakten rezultat neničelen z eksponentom večjim od največje možne vrednosti. Ob prekoračitvi se vrne največje še predstavljivo število ali neskončnost, oboje z ustreznim predznakom.

Do podliva pride, kadar je eksaktni rezultat neničelno število, ki ima po zaokroževanju (ob predpostavki, da je eksponent neomejen) eksponent manjši od najmanjše predstavljive vrednosti. (V primeru zaokroževanja k najbližjemu, je pat situacija zaokrožena k ničli.)

To pa ni edina možna definicija podliva. Obstaja tudi podliv pred zaokroževanjem. Do njega pride, kadar je eksponent manjši od najmanjše predstavljive vrednosti pri eksaktnem rezultatu. Do različnih vrst podlivov na primer pride pri majhnem pozitivnem številu in zaokroževanju navzgor.

Ob podlivu lahko implementacija vrne 0 ali pozitivno/negativno število z najmanjšo absolutno vrednostjo.

Do deljenja z ničlo pride, ko neničelno število delimo z ničlo. Vrnjena vrednost je pozitivna ali negativna neskončnost.

Napačna operacija se zgodi, kadar funkcija na podanem argumentu ni definirana, na primer kvadratni koren negativnega števila.

Neeksakten rezultat se zgodi vsakič, ko je rezultat zaokrožen. Lahko si predstavljamo, da je ta izjema precej pogosta in jo zato ponavadi ne želimo obravnavati s prekinitvami.

2.10 Povratne pretvorbe

Pri pretvorbi iz ene številske osnove v drugo moramo (razen v posebnih primerih) prav tako zaokroževati. Pretvorbe želimo predvsem pri vnosu in izpisu števil, saj nam je desetiška osnova veliko bolj domača, kot dvojiška, ki se uporablja interno.

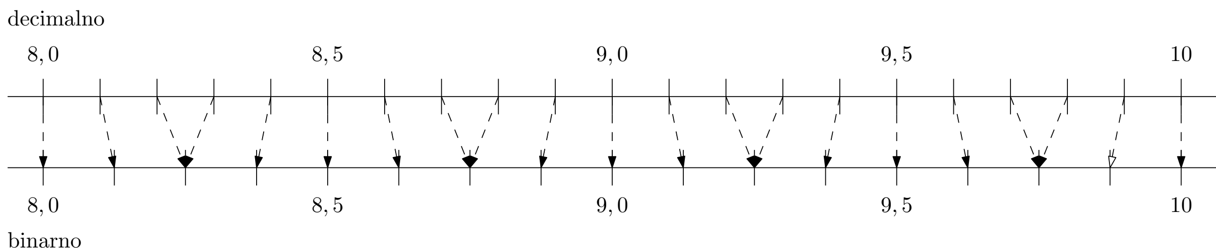
Pretvorbi pravimo, da je *povratna* (*angl. round-trip*), kadar nam pretvorba v drugo osnovo in obratno vrné isto število. Na primer pretvorba iz desetiškega sistema v dvojiškega in obratno.

Potreben pogoj, da bodo pretvorbe povratne je, da lahko v drugi osnovi izrazimo vsaj toliko števil kot v prvi. Kot primer si oglejmo pretvorbo iz desetiške osnove z natančnostjo dveh mest v dvojiško osnovo. V desetiškem sistemu lahko predstavimo 100 različnih mantis. V dvojiški osnovi tako potrebujemo vsaj 7 bitov, kar je 128 različnih mantis. To

deluje za cela števila, medtem ko moramo pri plavajoči vejici dodatno upoštevati različen razmik med števili.

Razmik med števili se pri desetiški osnovi spremeni pri vsaki potenci 10, medtem ko se pri dvojiški pri potencah števila 2. Sprememba ene in druge potence nam razdeli števila v območja z različnimi razmiki in posledično različnim številom predstavljivih števil. Da bodo pretvorbe povratne, mora biti v vsakem od teh območij dovolj števil v drugi osnovi.

Pri omenjenem primeru je dvojiških števil premalo v območju med 8 in 10, kot je prikazano na sliki 2.4. Zato 7 bitov ni dovolj. Se pa izkaže, da je 8 bitov dovolj.



Slika 2.4: Primer pretvorbe iz 2-mestnega desetiškega sistema v 7-mestni dvojiški sistem v območju med 8 in 10. Iz slike je razvidno, da se štirje pari števil pretvorijo v eno število in zato pretvorba ni povratna.

Za pretvorbe iz dvojiške osnove v osnovo b v večini primerov zadošča najmanjša natančnost m, izražena kot:

$$m = 1 + \left\lceil p \frac{\log 2}{\log b} \right\rceil ,$$

kjer je p natančnost števila, ki ga pretvarjamo. Kadar je b potenca 2, lahko nadomestimo p s $p-1$. Obstajajo pa zelo redki primeri, ko potrebujemo eno dodatno mesto. Za osnove do 62 je najmanjši tak primer, ko je $p = 186\,564\,318\,007$ v osnovah 7 in 49 [**?**].

Obratno za pretvorbe iz desetiške osnove v dvojiško velja:

$$m = \left\lfloor (b-1) \frac{\log 2}{\log 10} \right\rfloor .$$

Za oblike določene v IEEE 754 tako velja:

- desetiški zapis s 15 števkami (oziroma 6) lahko povratno pretvorimo v dvojno (oziroma enojno) natančnost,
- dvojno (oziroma enojno) natančnost lahko povratno pretvorimo v 17 desetiških števk (oziroma 9).

2.11 Knjižnice

Programske knjižnice uporabljamo, ker strojna podpora plavajoči vejici s poljubno natančnostjo ni razširjena. Na voljo je več knjižnic, ki podpirajo plavajočo vejico v poljubni natančnosti: MP, GMP, CLN in MPFR. Poleg tega obstajajo tudi računalniški programi, ki imajo implementacijo, vendar ni dostopna v obliki knjižnice. Primer so: Maple, Mathematica ali PARI/GP.

Knjižnice večinoma implementirajo osnovne aritmetične operacije, vendar se razlikujejo v implementaciji: programskem jeziku, interni številski osnovi in obliki števil, ostalimi matematičnimi funkcijami, ki so na voljo ter predvsem po učinkovitosti uporabljenih algoritmov in implementacije.

Problem z večino teh knjižnic je, da nimajo jasne semantike ali rezultati operacij niso pravilno zaokroženi. Izjema sta knjižnici GMP in MPFR.

Poleg pravilnega zaokroževanja in osnovne aritmetike želimo, da knjižnica podpira tudi druge matematične funkcije in da so operacije čim bolj učinkovite.

MPFR podpira navedeno [?] in ima poleg tega tudi dobre rezultate v primerjavah hitrosti različnih knjižnic.

3 Čisti funkcijski jeziki

V funkcijskih jezikih so funkcije enakovredne drugim vrednostim (*angl. first-class citizen*), kar pomeni, da jih podajamo kot argumente in vračamo kot rezultate drugih funkcij.

Jezik je *čist* (*angl. pure*), če se funkcije vedejo enako kot vrednosti oziroma nimajo *stranskih učinkov* (*angl. side effects*) ali *efektov*, kjer kot stranski učinek definiramo vsak programski konstrukt, ki lahko spremeni vrednost funkcije pri danih argumentih (npr. sprememba globalne spremenljivke) ali kako drugače vpliva na okolje. Stranski učinki so tudi izpisovanje na zaslon, prekinitve izvajanja (izjeme) in nedeterministična izbira ene od večih možnosti. Funkcije brez stranskih učinkov tako pri fiksnih vrednostih argumentov vrnejo vedno enak rezultat, enako kot so definirane v matematiki

Kako lahko potem v takih jezikih sploh kakorkoli vplivamo na okolje ali na primer implementiramo funkcijo, ki bo vrnila trenuten čas? Možna rešitev je, da vsako tako funkcijo opremimo s parametrom, ki opisuje trenutno stanje sveta (`State# RealWorld`). Ker se stanje sveta vedno spreminja, lahko tudi funkcija vsakič vrne trenutni čas in ostane čista. Ker se je med klicem stanje spremenilo, je najlažje da taka funkcija vrne tudi novo stanje sveta.

Ker bi bilo tako podajanje stanja med klici funkcij zelo zamudno, uporabimo koncept *monade*. Monada poskrbi za to, da za nas vodi „evidenco" o stanju sveta. To stori tako, da definira kaj se zgodi med dvema zaporednima klicema. Vse operacije z efekti morajo potekati znotraj monade, da jezik ostane čist.

3.1 Neučakani in leni jeziki

Predvsem funkcijske jezike lahko, glede na zaporedje v katerem se izrazi evaluirajo, v grobem delimo na *neučakane* (*angl. eager*) in *lene* (*angl. lazy*).

Večina jezikov (predvsem ukaznih) je neučakanih. Pri neučakanih jezikih se pri aplikaciji $e_1 e_2$ vedno najprej evaluirata izraza e_1 in e_2 ter nato aplikacija [?]. Z drugimi besedami se vrednost izračuna takoj, ko je prirejena spremenljivki ali podana kot argument. Podatkovne strukture so zato evaluirane v celoti.

Pri lenih jezikih se v aplikaciji evaluira le izraz e_1. Izraz e_2 se evaluira šele, ko ga telo funkcije potrebuje [?]. Podatkovne strukture so lahko posledično le delno evaluirane.

To je sicer zelo groba delitev jezikov, saj v večini jezikov poznamo konstrukte, ki se evaluirajo na drugačen način.

Neučakano evaluacijo je lažje učinkovito implementirati, saj imamo zanjo dobro strojno podporo. Prednost je tudi, da je uporabniku sledenje evaluaciji lažje, sploh če so v jeziku prisotni stranski učinki. Slabost neučakane evaluacije je, da lahko pride do nepotrebnih evaluacij.

Prednost lenih jezikov je ta, da lahko operiramo tudi z izrazi, ki v neučakanem jeziku ne bi imeli pomena. Primer tega so neskončne podatkovne strukture. V lenih jezikih lahko brez težav definiramo neskončne sezname in na njih uporabimo smiselne operacije (npr. operacija ki vrne glavo ali rep takega sezname je smiselna, medtem ko dolžina seznama ni). Ker je evaluacija lena, se bo evaluiral le tisti del strukture, ki ga potrebujemo, da izračunamo rezultat.

3.2 Implementacija – GHC

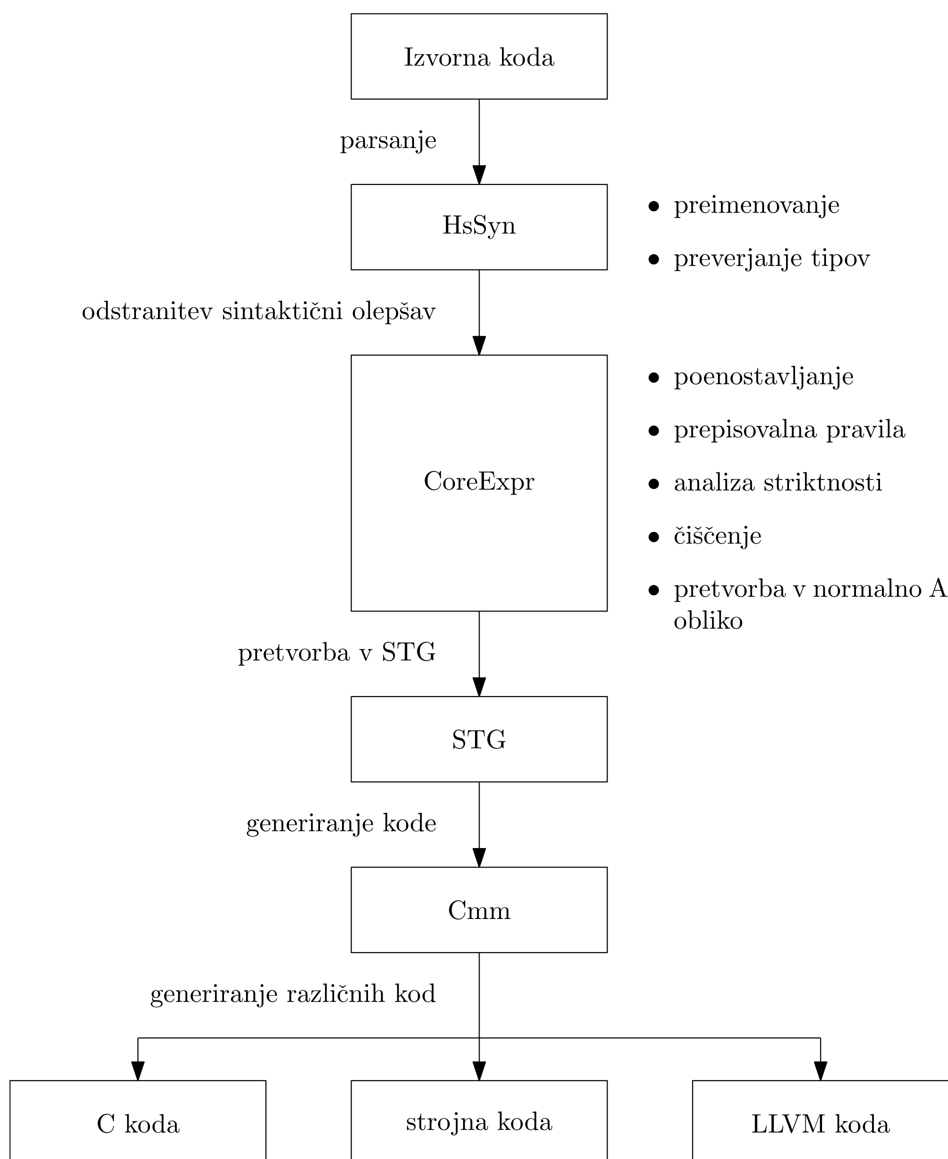
Ker je jezik len, se neizvedeni deli kode zapišejo v pomnilniku kot *grude* (*angl. thunk*). Ko jih potrebujemo se evaluirajo oziroma izvedejo in gruda se nadomesti z vrednostjo (*angl. thaw*). Tako kot koda so lahko vrednosti izrazov le delno evaluirane.

3.2.1 Primitivni tipi in operacije

Nove tipe v Haskell-u iz že obstoječih tvorimo s pomočjo produkta, disjunktne vsote in funkcijskega prostora. Pri tem produkt ali vsak člen vsote poimenujemo s konstruktorjem. Taki tipi se zaradi konstruktorja imenujejo *uokvirjeni* (*angl. boxed*). Tudi osnovni tipi, ki jih imamo na razpolago, so uokvirjeni, kot na primer: cela števila, znaki in resničnostne vrednosti.

Za učinkovito izvajanje potrebujemo poleg uokvirjenih tudi *neuokvirjene* (*angl. unboxed*) tipe. Neuokvirjeni tipi predstavljajo strojne vrednosti in sledijo neučakanemu modelu evaluacije. Neuokvirjeni tipi so del interne implementacije in zato uporaba ni zaželena, razen v posebnih primerih. Da so jasno ločeni od uokvirjenih tipov, se njihova imena končajo z lestvico (npr. `Int#`, `Char#`). Lestvico uporabimo tudi za označevanje funkcij, ki uporabljajo neuokvirjene tipe, in primitivne oziroma strojne operacije (npr. `+#`, `*#`).

Prednost neuokvirjenih tipov je, da ne zasedejo toliko prostora v pomnilniku kot uokvirjeni, saj se pri njih ne tvorijo grude. Neuokvirjene tipe lahko prevajalnik optimizira in jih lahko hrani v registrih, medtem ko uokvirjenih ne more.



Slika 3.1: Stopnje GHC prevajalnika

Navedimo omejitve neuokvirjenih tipov [?], ki pa za razumevanje potrebujejo poznavanje Haskella. Neuokvirjeni tipi ne morejo biti uporabljeni v polimorfnih funkcijah. Ko tvorimo nove tipe, jih ne moremo uporabiti na najvišjem nivoju. Prav tako jih ne moremo uporabljati v rekurzivnih klicih. V vzorcih jih lahko uporabimo le, kadar so le-ti striktni.

Iz neuokvirjenih tipov lahko sestavimo produktni tip, tako imenovane *neuokvirjene n-terice*, ki se zapišejo v oklepajih z lestvico, na primer: `(# Int#, Int# #)`. Enako zapišemo tudi striktni vzorec za neuokvirjene n-terice.

Neuokvirjene n-terice so uporabne za funkcije, ki vračajo več vrednosti hkrati. Zaradi pogostosti tega primera v funkcijskih jezikih, je operacija v prevajalniku optimizirana in se vrednosti vrnejo v več registrih. To je bolje od običajnih dogovorov o klicih, ki predvidevajo samo eno vrnjeno vrednost.

3.2.2 Uporaba kode iz drugih jezikov

V ta namen služi *vmesnik za tuje funkcije* (*angl. foreign function interface*). Osnovna uporaba za klice C-jevih funkcij je enostavna. Navedemo originalno ime in signaturo funkcije ter ime, ki ga želimo uvoziti v Haskell. Prevajalnik pri tem poskrbi za vse, kar je potrebno, da se funkcije pravilno pokličejo. Poskrbi za *razporejanje argumentov* (*angl. marshalling*) iz Haskell-ovih tipov v C-jeve in upošteva dogovore, ki veljajo za take klice (*angl. calling convention*).

Razporejanje osnovnih tipov je že definirano, medtem ko lahko razporejanje za bolj kompleksne tipe definiramo sami. Pazljivost pri dodelitvi pomnilnika in pri delu s kazalci je nujna. Ker so dodelitve pomnilnika stranski učinek, jih je treba uporabljati v monadi. Samo razporejanje ni najbolj učinkovito in posledično tudi klici pri katerih je le-to potrebno.

Učinkovitejša je uporaba neuokvirjenih tipov, s katero se lahko izognemo razporejanju oziroma ga implementiramo sami. Na ta način lahko napišemo tudi klice funkcij napisane v jeziku Cmm.

3.2.3 C-- in Cmm

C-- je na prvi pogled okrnjena različica C-ja. Služi kot vmesna stopnja prevajalnikov višje-nivojskih jezikov, kot so na primer funkcijski jeziki. Zapisan je v človeku berljivi obliki, kar je sicer neobičajno za vmesne stopnje. Gre za ukazni jezik (*angl. imperative language*), kot je C.

Jezik pozna le dve vrsti tipov, ki sledijo strojni opremi. Prva vrsta so bitni tipi, ki imajo lahko poljubno, vendar vnaprej določeno dolžino (`bits8`, `bits16`, `bits32`, `bits64`). Spremenljivke tega tipa so primerne za registre. Kakšne vrednosti predstavljajo (predznačena ali nepredznačena cela števila, števila v plavajoči vejici, kazalce) določajo operacije, ki so na njih uporabljene. Poleg tega obstaja še dvojiška vrednost (*angl. boolean*), ki jo lahko uporabimo le v stavkih, ki določajo potek izvajanja operacij. Število spremenljivk ni omejeno in naloga prevajalnika je, da jih priredi registrom. Zaradi tega je jezik bistveno bolj berljiv.

Na voljo so nam primitivne operacije, ki jih sestavljajo: bitne operacije, pretvorbe, aritmetične operacije (na celih številih in številih v plavajoči vejici) in testi. Le-te predstavljajo univerzalni zbirni jezik. Funkcije lahko za razliko od C-ja vračajo tudi več kot eno vrednost, kar v C-ju ni tako enostavno, medtem ko zbirnemu jeziku to ne predstavlja večjih težav. Kot kontrolni stavki so nam na voljo `if`, `switch` in `goto`, vendar so za slednjega dovoljeni le skoki znotraj funkcij.

Ni pa vse samo okrnjeno v primerjavi s C-jem. Na voljo imamo poleg običajnega klica procedur tudi klice, ki omogočajo učinkovito in elegantno implementacijo enostavne rekurzije (*angl. tail recursion*) ter klice, s katerimi lahko naravno implementiramo kontinuacije. Poleg tega C-- izpostavi svojo knjižnico, ki omogoči, da v C-ju implementiramo dele, ki jih potrebujejo višje-nivojski jeziki. Ti so na primer odstranjevanje smeti (*angl. garbage collection*) in obravnavanje izjem (*angl. exception handling*).

GHC uporablja malenkost prirejeno različico C-- imenovano Cmm. Razlike so minimalne. Cmm predstavlja zadnjo stopnjo prevajalnika, preden se proces razveji v več različnih produktov (slika 3.1).

3.2.4 Profiliranje

Potek izvajanja programov v čistem in lenem funkcijskem jeziku je precej drugačen od ukaznega stila, ki ga podpira strojna oprema, zato je tudi profiliranje drugačno. Da lahko program profiliramo, ga moramo prevesti z dodatnimi stikali, prav tako vse knjižnice, ki jih uporabi. Ta stikala spremenijo glave pomnilniških blokov, tako da dodajo prostor za dodane informacije o izrazu, ki se je izvedel.

3.2.5 Upravljanje s pomnilnikom in odstranjevanje smeti

Večina vrednosti je v Haskell-u nespremenljivih (*angl. immutable*), zato posledično nastane bistveno več smeti. Vsaka operacija namreč ustvari novo vrednost. Ni neobičajno, da programi ustvarijo več kot 1 GB podatkov v sekundi, vendar se večina tudi takoj odstrani [?].

Glavna lastnost nespremenljivih vrednosti je v tem, da nikoli ne kažejo na mlajše vrednosti. Mlajše vrednosti namreč še ne obstajajo, hkrati starejših ne moremo več spremeniti.

Ta lastnost poenostavi nalogo odstranjevanja smeti. Kadarkoli lahko preletimo zadnje nastale vrednosti in tiste, na katere ni nobenega kazalca, takoj odstranimo.

Zadnje vrednosti se ustvarijo v 512 kB veliki negovalnici (*angl. nursery*). Ko je leta v celoti izkoriščena, se zgodi manjše odstranjevanje smeti. Vendar namesto, da bi odstranjevali nepotrebne vrednosti, je učinkoviteje, da še potrebne vrednosti, prestavimo v glavni del pomnilnika.

Programi tako delujejo hitreje, kadar je večji del njihovih vrednosti primeren za odstranitev.

3.2.6 Predstavitev literalov

Literal je fiksna vrednost, ki jo podamo v izvorni kodi. Zanimajo nas literali celih števil in ulomkov. Posebno pozornost literalom posvečamo zato, ker je za pretvorbo števila v plavajoči vejici v interno obliko običajno potrebna operacija (razen če so literali podani v dvojiški ali šestnajstiški obliki z dvojiškim eksponentom).

Sprememb tipov (*angl. type cast*) se v Haskell-u poskusimo v največji meri izogniti. Tako dobijo celoštevilski literali tip `Num a => a` in literali, ki so zapisani s plavajočo vejico, `Fractional a => a`. Tip literala je določen le do razreda in dejanski tip dobi šele, ko ga uporabimo. Na primer tip izraza `6 * 7 :: Num a => a` je splošen, dokler ne zahtevamo rezultata `6 * 7 :: Int` (tip `Int` je v Haskellu 32 ali 64 bitno celo število). Takrat se bodo izvedle pretvorbe literalov 6 in 7 v potreben tip, tako da se pokliče ena izmed funkcij `fromInteger` (v tem primeru definirana v instanci `instance Num Int`).

Razred `Num` poleg ostalih vsebuje funkcijo `fromInteger :: Integer -> a`, medtem ko razred `Fractional` vsebuje `fromRational :: Rational -> a`, kjer je tip `Rational` definiran kot par dveh celih števil s poljubno natančnostjo, ki predstavlja ulomek. Za učinkovito implementacijo moramo torej zagotoviti ti dve funkciji.

4 Uporaba knjižnice MPFR v GHC-ju

Pri prenosu MPFR knjižnice sledimo prenosu knjižnice GMP, ki je v GHC-ju uporabljena kot osnova za poljubno natančna cela števila. Računamo na to, da je bilo pri slednjem vloženo veliko truda, da je prenos učinkovit. Dodatna prednost je, da je MPFR narejena na osnovi GMP knjižnice.

Prenos temelji na knjižnici Rounded[1] avtorja Edwarda Kmetta. Osnovna ideja je enaka, vendar so v Kmettovem vmesniku uporabljeni tipi višjega reda parametrizirani z natančnostjo in načinom zaokroževanja. To sicer omogoči implementacijo standardnih Haskellovih razredov, ki ne vsebujejo natančnosti. Vendar taki tipi močno otežijo uporabo, kadar želimo med izračunom spreminjati natančnost.

To pa ni edini poskus vmesnika za MPFR v Haskell-u. Predhodni poskus s knjižnico HMPFR[2] je naredil Aleš Bizjak. Knjižnica trpi zaradi občasnih, vendar nepredvidljivih zaustavitev delovanja. Predlagana rešitev je, da GHC prevedemo s preimenovano knjižnico GMP ali z alternativno knjižnico GMP imenovano `integer-simple`. Enako je potrebno potem storiti tudi za programe, ki uporabljajo knjižnico, če so le-ti dinamično povezani.

4.1 MPFR

Knjižnica MPFR (*angl. Multiple Precision Floating-Point Reliable Library*) je napisana v C-ju. Cilj njene implementacije je natančna semantika. Glavne lastnosti, ki jo ločijo od večine drugih knjižnic, so:

- prenosljivost kode (rezultati operacij niso odvisni od velikosti strojne besede),
- natančnost je podana v bitih in se tako tudi upošteva (vključno pri majhnih natančnostih) ter
- ima 5 načinov zaokroževanja tako za aritmetične operacije kot za ostale matematične funkcije.

[1] `https://github.com/ekmett/rounded`
[2] `https://hackage.haskell.org/package/hmpfr`

Če uporabimo natančnost 53-ih bitov, potem MPFR natančno reproducira izračune narejene s strojno dvojno natančnostjo. Razlika je le v večjem obsegu eksponenta in tem, da subnormalna števila niso implementirana (lahko pa jih emuliramo).

Knjižnica je od verzije 3 naprej dostopna pod GNU LGPL licenco. Lahko jo vključimo tudi v programe, ki niso dostopni brezplačno, le da zraven priložimo izvorno kodo MPFR in možnost, da se program uporabi s spremenjeno knjižnico.

4.1.1 Zaokroževanje

MPFR podpira vse načine zaokroževanja, ki so definirani v IEEE 754 standardu in še enega dodatnega, t. j. zaokroževanje stran od ničle.

Večina funkcij v MPFR knjižnici ima prvi parameter kazalec na spremenljivko, kamor se zapiše rezultat. Sledijo vhodni parametri in kot zadnji parameter je način zaokroževanja. Funkcije vračajo celoštevilsko vrednost, ki se imenuje tudi ternarna vrednost.

Kadar je vrnjena ternarna vrednost enaka nič, je rezultat eksakten (ni bil zaokrožen). V primeru pozitivne (oz. negativne) ternarne vrednosti je vrnjeno število večje (oz. manjše) od eksaktnega rezultata. V primeru, da je rezultat neskončnost, do katere je prišlo zaradi preliva, se rezultat obravnava kot ne-eksakten. Rezultat NaN je vedno obravnavan kot eksakten.

4.1.2 Rezultat na posebnih številih

Funkcije imajo lahko enega ali več vhodnih parametrov. Kadar noben od argumentov ni enak NaN, lahko na vhodne argumente gledamo kot na n-terico razširjenih realnih števil (realna števila in pozitivna ter negativna neskončnost).

Kadar je funkcija definirana na razširjenih realnih številih, je rezultat pravilno zaokrožena vrednost (posebna pravila veljajo za predznačeno ničlo).

Kadar funkcija ni definirana na razširjenih realnih številih:

1. Funkcijo je mogoče zvezno razširiti: rezultat je ustrezna limita.
   Na primer: `mpfr_hypot`[3]`(+Inf,0)=+Inf`. Funkcije `mpfr_pow` s tem pravilom ne moremo razširiti na $(1,\infty)$.

2. Kadar je eden od argumentov $+0$ (oz. -0) in funkcija ni definirana na 0:

   (a) vendar je razširljiva na 0 – rezultat je ustrezna limita, ko se vhodni argument približuje 0 od zgoraj (oz. od spodaj),

   (b) limita ne obstaja – rezultat je opisan pri vsaki funkciji posebej.

[3]`mpfr_hypot(x,y)=`$\sqrt{x^2+y^2}$

3. Kadar rezultata ne moremo definirati niti z limito, je vrnjena vrednost NaN.

Kadar je rezultat enak 0 je predznak določen, kot da za vhodni argument funkcija ne bi bila definirana.

Kadar je eden od vhodnih argumentov NaN, je rezultat NaN, razen v primeru, da je funkcija v tem parametru konstantna na vseh končnih številih. Na primer: `mpfr_hypot(NaN,0) = NaN` in `mpfr_hypot(NaN,+Inf) = +Inf`, saj je za katerikoli končni x `mpfr_hypot(x,+Inf) = +Inf`.

4.1.3 Izjeme

MPFR ima 6 vrst izjem: podliv, prekoračitev, deljenje z ničlo, NaN, ne-eksakten rezultat in napaka območja (*angl. range error*).

Podliv, prekoračitev in deljenje z ničlo smo že opisali v poglavju 2.9.

Izjema „ne-ekstaken rezultat" se zgodi, ko je rezultat funkcije zaokrožen.

Napaka območja se lahko zgodi le pri funkcijah, ki ne vračajo števila v plavajoči vejici. Recimo pri primerjavah ali pri pretvorbi v cela števila.

MPFR ima za vsako izjemo globalno zastavico, ki jo lahko počistimo, nastavimo ali preverimo.

NaN se zgodi vsakič, ko je rezultat funkcije NaN.

Razlike v primerjavi z ISO C99 standardom:

- V C-ju je definiran le tihi NaN, ki se propagira in ne proži izjem. Razen v posebnih primerih, MPFR vedno nastavi NaN zastavico, kadar je rezultat NaN (tudi kadar se le propagira), kot da bi bil vsak NaN signalni.
- Neveljavna operacija v C-ju (*angl. invalid exception*) ustreza bodisi izjemi NaN bodisi napaki območja v MPFR.

4.1.4 Upravljanje s pomnilnikom

MPFR funkcije lahko shranijo vrednosti v lasten predpomnilnik (na primer pri izračunu konstant, kot je π). To se lahko zgodi, ko kličemo funkcijo za izračun konstante ali kadar je le-ta potrebna interno.

Kadarkoli lahko predpomnilnik sprostimo s klicem `mpfr_free_cache`. To je priporočeno storiti pred zaključkom níti ali pred izhodom iz programa (zaradi orodij, ki ugotavljajo puščanje pomnilnika (*angl. memory leak*)).

MPFR-jevi interni podatki, kot so zastavice, območje eksponenta, privzeta natančnost in način zaokroževanja ter predpomnilnik so lahko globalni (prevajanje z opcijo `thread safe`) ali lokalni za vsako nit (opcija `TLS`). Začetne vrednost spremenljivk v TLS po

kreaciji nove niti so popolnoma odvisne od prevajalnika in implementacije niti (MPFR sicer upošteva dogovore o inicializaciji in deklaraciji glede na implementacijo niti).

4.1.5 Nomenklatura in tipi

C-jev tip, ki v MPFR-ju predstavlja število v plavajoči vejici, je `mpfr_t`. Interno je definiran kot struktura, ki vsebuje natančnost, predznak, eksponent ter vektor, ki predstavlja mantiso (primer 4.3).

Zavzame lahko tri posebne vrednosti: NaN, pozitivno in negativno neskončnost. NaN je začetna vrednost vseh spremenljivk, rezultat nedovoljene operacije (deljenje z ničlo) ali neznane vrednosti (kot $\infty - \infty$). Enako kot v IEEE 754 standardu je ničla predznačena in se obnaša enako. Posplošena je tudi na druge funkcije, ki jih podpira MPFR.

Natančnost je število bitov mantise in ustrezen tip v C-ju je `mpfr_prec_t`. Izražena je lahko s katerimkoli celim številom med `MPFR_PREC_MIN` in `MPFR_PREC_MAX`. V trenutni implementaciji je `MPFR_PREC_MIN` enak 2.

MPFR interno potrebuje večjo natančnost, da lahko vrača pravilno zaokrožene rezultate. Zato ni priporočljiva uporaba natančnosti v bližini vrednosti `MPFR_PREC_MAX`. V kolikor se interno natančnost prekorači, MPFR prekine delovanje programa (*angl. assertion failure*). Tudi v primeru, ko dosežemo določene omejitve pri porabi pomnilnika, se lahko program obesi, prekine ali ima nedefinirano vedenje (odvisno od implementacije C-ja).

Način zaokroževanja je definiran s tipom `mpfr_rnd_t`.

4.1.6 Dogovori o spremenljivkah

Preden MPFR spremenljivki priredimo vrednost, je spremenljivko potrebno inicializirati z eno od temu namenjenih funkcij in jo po uporabi prav tako sprostiti. Spremenljivko je potrebno inicializirati samo enkrat, medtem ko ji lahko rezultat priredimo poljubno krat. Zaradi učinkovitosti, je bolje da se izognemo inicializaciji in sprostitvi spremenljivk v zankah in to storimo pred in za zanko. Skrb za dodelitev dodatnega pomnilnika ni potrebna, saj je velikost porabljenega pomnilnika fiksna – odvisna od natančnosti. Tako bo spremenljivka, razen če spremenimo natančnost ali jo sprostimo in ponovno inicializiramo, ves svoj obstoj porabila enako pomnilnika.

4.1.7 Funkcije knjižnice

Funkcije knjižnice MPFR lahko razdelimo v naslednje skupine:

- inicializacija in prirejanje,

- pretvorbe,
- osnovna aritmetika,
- primerjave,
- transcendentne funkcije in
- ostale funkcije.

4.2 Funkcijski vmesnik

C-jev vmesnik knjižnice MPFR je izrazito ukazen z namenom, da bi bil čim bolj učinkovit. Vsako spremenljivko moramo ročno inicializirati: ji nastaviti natančnost, kar rezervira prostor v pomnilniku in ji nastaviti začetno vrednost. Parametri vseh operacij so nato kazalci na te spremenljivke. Na koncu moramo zopet ročno poskrbeti, da se pomnilnik tudi sprosti.

Primer 4.1 Primer uporabe knjižnice MPFR v C-ju.

```
#include <stdio.h>

#include <gmp.h>
#include <mpfr.h>

int main (void)
{
  unsigned int i;
  mpfr_t s, t, u;

  mpfr_init2 (t, 200);
  mpfr_set_d (t, 1.0, MPFR_RNDD);
  mpfr_init2 (s, 200);
  mpfr_set_d (s, 1.0, MPFR_RNDD);
  mpfr_init2 (u, 200);
  for (i = 1; i <= 100; i++)
    {
      mpfr_mul_ui (t, t, i, MPFR_RNDU);
      mpfr_set_d (u, 1.0, MPFR_RNDD);
      mpfr_div (u, u, t, MPFR_RNDD);
      mpfr_add (s, s, u, MPFR_RNDD);
    }
  printf ("Sum is ");
  mpfr_out_str (stdout, 10, 0, s, MPFR_RNDD);
  putchar ('\n');
  mpfr_clear (s);
  mpfr_clear (t);
  mpfr_clear (u);
  return 0;
}
```

Tak način klicev je v Haskell-u sicer izvedljiv, vendar ni intuitiven za funkcijsko programiranje. Zato Haskell-ov vmesnik prilagodimo funkcijskemu programiranju. Inicializacijo združimo z ostalimi funkcijami, tako da vračajo že inicializirane vrednosti. Odstranjevalcu smeti prepustimo, da upravlja s pomnilnikom.

Primer 4.2 Primer uporabe knjižnice MPFR v Haskell-u.

```
import Data.Approximate.MPFRLowLevel

let sum 0 = (fromDouble Down 200 1, fromDouble Down 200 1)

let sum n =
    let (s',t') = sum (n-1) in
      (add Down 200 s' div', mul Up 200 t' i)
    where
      one = fromDouble Down 200 1
      div' = div Down 200 one t'
      i = fromIntegerA Down 200 n

let main =
   print $ "Sum is " ++ (toString Down 10 sum')
   where (sum',_) = sum 100
```

4.3 Predstavitev MPFR vrednosti v GHC

Interno MPFR predstavlja svoje vrednosti v strukturi, ki vsebuje natančnost, predznak, eksponent in niz, ki predstavlja mantiso (primer 4.3). Struktura se pri uporabi alocira na skladu, medtem ko se mantisa alocira na kopici.

Primer 4.3 Struktura osnovnega tipa MPFR-jevih števil v plavajoči vejici v C-ju.

```
typedef int mpfr_prec_t;    /* Definition of precision */

typedef int mpfr_sign_t; /* Definition of sign */

typedef int mpfr_exp_t; /* Definition of exponent */

/* Definition of the main structure */
typedef struct {
  mpfr_prec_t   _mpfr_prec;
  mpfr_sign_t   _mpfr_sign;
  mpfr_exp_t    _mpfr_exp;
  mp_limb_t    *_mpfr_d;
} __mpfr_struct;

typedef __mpfr_struct mpfr_t[1];
```

V GHC-ju to predstavimo s podobno strukturo (primer 4.4). Zaradi učinkovitosti združimo predznak in natančnost, saj se bodo tako pri klicih za vrednosti bolj pogosto

uporabili registri. Za mantiso uporabimo interni tip `ByteArray#` za katerega skrbi GHC-jev odstranjevalec smeti.

Primer 4.4 Struktura osnovnega tipa MPFR-jevih števil v plavajoči vejici v Haskell-u.

```haskell
{- Basic data -}
type CPrec#       = Int#
type CSignPrec#   = Int# -- Sign# << 64/32 | Precision#
type CPrecision# = Int#
type CExp#        = Int#
type CRounding#   = Int#

data Rounded = Rounded
  { roundedSignPrec :: CSignPrec#
  , roundedExp      :: CExp#
  , roundedLimbs    :: ByteArray#
  }
```

4.4 Prenos funkcij z enako signaturo

Pri implementaciji vmesnika v Haskell se soočimo z velikim številom funkcij s približno enako signaturo. Medtem ko pri industrijskem programiranju ni neobičajno, da se del kode ponavlja, se temu vendarle poskušamo izogniti. Ponavljanje kode ni le monotono opravilo, ampak se s tem zmanjša tudi kvaliteta kode.

Idealna rešitev bi bila uporaba Haskell-ovih predlog (*angl. Template Haskell*). Prevajalnik v prvi fazi iz izvorne kode izdela sintaktično drevo. Predloge nam omogočajo, da že med samim prevajanjem posežemo v to drevo. Del kode se prevede in izvede. Ta del kode izdela novo sintaktično drevo, ki je potem uporabljeno za končni program.

Nastopi težava, da v verziji GHC-ja 7.8, sintaktično drevo, ki je dostopno predlogam, ni v celoti podprto. Manjkajo klici kode napisane v Cmm.

Poleg predlog Haskell podpira uporabo različnih predprocesiranj kode (*angl. preprocessor*). Predprocesiranje kode, deluje bolj ali manj na tekstovni osnovi izvorne kode. Podprto je C-jevo predprocesiranje, kot tudi drugi načini predprocesiranja.

Z uporabo predprocesiranja izgubimo možnost preverjanja sintaktične pravilnosti že v prvi fazi. Tudi koda ni več tako berljiva.

Z uporabo C-jevih makrov rešimo dva problema. Z makri opišemo dober del funkcij s približno desetimi različnimi signaturami. Ena definicija makra poskrbi za kodo v Cmm-ju, medtem ko druga definicija za uvožene funkcije v Haskell-u.

4.5 Funkcije za pretvorbo in literali

Za razliko od operacijami nad števili, pa imajo literali in pretvorbe popolnoma specifične signature. Prenesti moramo vsako posebej, pri tem pa upoštevamo specifike literalov v Haskellu. Tako za literale implementiramo funkcije `fromIntegerA` in `fromRationalA`, ki kot vhod sprejmejo neomejeno celo število oziroma ulomek ter željeno natančnost in način zaokroževanja. Ravno zaradi dodatnih parametrov, ne moremo podpreti standarnih Haskellovih razredov.

Enako velja tudi za pretvorbe v in iz nizov pri kateri tudi potrebujemo natančnost in način zaokroževanja, saj pri pretvorbi med dvojiško in desetiško osnovo pride do zaokto-ževanja. Izognemo se C-jevmu načinu branja in izpisovanja, ki uporablja vzorce in vsak vzorec prenesemo posebaj: `toStringHex`, `toStringBin`, `toStringSci`, `toStringFix` vse s signaturo `RoundMode -> Int -> Rounded -> String`, obenem pa pustimo na voljo bolj splošno funkcijo `toRawStringExp`, ki kot dodaten parameter sprejme številsko osnovo. Omenimo še funcijo `toStringReadback :: Rounded -> String`, ki uporabi natačnost potrebno za povratno pretvorbo opisano v poglavju 2.10.

4.6 Uporaba Cmm in neuokvirjenih tipov

Uporabo Cmm in neuokvirjenih tipov najlažje razložimo s primerom poteka klica ene funkcije. Oglejmo si funkcijo

```
add ::  RoundMode -> Precision -> Rounded -> Rounded -> Rounded.
```

Taka signatura je skupna vsem binarnim funkcijam. Prva dva parametra predstavljata način zaokroževanja in natančnost rezultata. Sledita števili, ki sta vhod operacije.

Prvi koraki v klicu se izvedejo v Haskell-ovi kodi, in sicer se vsi uokvirjeni tipi pretvorijo v neuokvirjene. Način zaokroževanja, ki so v Haskellu podani kot enumeracija, se pretvorijo v neuokvirjeno celo število, ki ustreza C-jevi enumeraciji. Natančnost se pretvori v neuokvirjeno celo število. Vhodni števili pa se iz strukture pretvorita v tri parametre za vsako število: predznak z natančnostjo, eksponent in mantiso.

S temi parametri se kliče implementacija v Cmm-ju `mpfr_cmm_add(W_ rnd, W_ prec, W_ ps1, W_ exp1, P_ limbs1, W_ ps2, W_ exp2, P_ limbs2)`. Za klic potrebujemo vmesnik za tuje funkcije (FFI).

Implementacija v Cmm-ju poskrbi, da se na skladu zgradijo MPFR-jevi C-jevi parametri. Na skladu inicializira tudi število, ki bo vrnjeno, na željeno natančnost s klicem

funkcije `mpfr_init2`. Izvede se klic `mpfr_add`. Vrnjeno število se razpakira, tako da je vrnjena neuokvirjen n-terica.

Klic se zaključi v Haskellovi kodi s pretvorbo slednje n-terice v uokvirjeno število.

4.7 Primerjava hitrosti

Učinkovitost MPFR-ja pogosto primerjajo z učinkovitostjo drugih knjižnic. V ta namen je izdelan preprost program, ki izmeri hitrost osnovnih operacij in nekaterih matematičnih funkcij. Ta program je nato prepisan tako, da uporabi vsako od knjižnic zajetih v primerjavo.

MPFR-jeva različica je napisana tako, da se dodelitev pomnilnika ne meri. To je tudi smiselno, saj v C-ju za dodeljevanje pomnilnika skrbi običajen klic `malloc`.

Napisali smo različico tudi za Haskell, ki uporablja čist funkcijski vmesnik. Tu se pri vsaki operaciji alocirajo nove spremenljivke, vendar se v ozadju uporablja interni klic Haskell-a. Dodali smo še test za običajno prirejanje, kar je poleg funkcijskega klica le dodelitev pomnilnika, in test seštevanja.

Rezultati primerjave so podani v tabeli 4.1. Pri enostavnih operacijah je Haskellov vmesnik manj kot 2 krat počasnejši. S kompleksnostjo operacij pa se razlika manjša. Tako je pri transcendentnih funkcijah hitrost vmesnika v Haskellu popolnoma primerljiv s C-jevim. Čas uporabljen za delo s pomnilnikom je zanemarljiv.

Pri nekaterih bolj zahtevnih funkcijah, nas preseneti, da Haskell doseže celo boljše čase kot C. Ker smo uporabili klice v C-jevo kodo in dodatno pretvarjanje argumentov, na hitrost ne more vplivati število inštrukcij. MPFR knjižnica je za GHC prilagojena tako, da namesto klicev za delo s pomnilnikom `malloc` in `free` iz standardne knjižnice uporablja GHC-jevo implementacijo. Le-ta dela linearno in je zelo učinkovito tako dodeljevanje kot tudi odstranjevanje pomnilnika. Zahtevne funkcije interno uporabljajo veliko začasnih spremenljivk in zato dosežejo tudi boljše čase. Meritev je sicer tekla dovolj časa, da se je vmes izvedel tudi GHC-jev odstranjevalec smeti, vendar bi ga morali, za nepristranski rezultat, tik pred koncem meritve še enkrat zagnati.

operacija	C [ms]	Haskell [ms]
prirejanje		0,0020
$x + y$		0,0029
$x \cdot y$	0,0030	0,0055
$x \cdot x$	0,0018	0,0055
x/y	0,0048	0,0069
$\sqrt{x}$	0,0048	0,0061
$\exp x$	0,17	0,17
$\log x$	0,16	0,16
$\sin x$	0,18	0,15
$\cos x$	0,16	0,15
$\arccos x$	0,38	0,32
$\arctan x$	0,36	0,32

Tabela 4.1: Primerjava hitrosti operacij v C-ju in v Haskell-u na `Intel Core i5-4200U`.

5 Zaključek

Čisti funkcijski jeziki, kot je Haskell, so zelo drugačni od klasičnih ukaznih jezikov, ki danes prevladujejo v uporabi. Razlika ni le v zasnovi jezika, ki od programerja zahteva drugačen način razmišljanja, ampak tudi v izvajanju programov. Zaradi razširjenosti ukaznih jezikov in tudi verjetno zato, ker so se v razvoju pojavili pred funkcijskimi jeziki, je strojna oprema prilagojena izvajanju le teh. Trenutno tako ni druge rešitve kot, da se izvajanje funkcijskih jezkov prilagodi temu, kar je na voljo.

Take prilagoditve so običajo uporabniku skrite. Haskellovi programe, ki ne posegajo v to, lahko prevedemo tako z GHC-jem, kot tudi s kako drugo implementacijo Haskellovega prevajalnika. Vmesnik za knjžnico sledi čistemu funkcijskemu slogu in na podoben način pred uporabnikom skrije postopke, ki so značilni za imperativni slog (npr. ročno upravljanje s pomnilnikom).

Da pa lahko to storimo, moramo poseči globoko v proces prevajalnika in prilagoditve na specifično implementacijo prevajalnika so nujne. Implemetacija prevajalnika je standardizirana le do neke mere. Bolj tudi ne more biti, saj odločitve in izbire pri implementaciji niso enostavne in so predmet raziskav. Do večjih razlik ne prihaja le med različnimi implementaciji, ampak tudi med samimi različicami GHC-ja.

To je tudi slabost našege prenosa knjžnice MPFR, saj je prilagojen eni različici GHC-ja. Ta prilagoditev močno oteži vzdrževanje knjižnice, saj jo moramo popraviti za vsako novo različico. Vendar tu nimamo dosti izbire. V primeru, da se ne bi odločili za specifično implementacijo GHC-ja, bi morali žrtvovati vmesnik knjižnice ali njeno hitrost.

Največ težav je bilo pričakovanih ravno pri hitrosti. Imperativni slog nam omogoča ročno upravljanje s pomnlinikom in enostavno si predstavljamo potek programa in klice v knjižnico. Haskell to naredi za nas in pričakovati je, da se bo čas izvajanja povečal za nekaj faktorjev. Z meritvami smo pokazali, da temu le ni tako in je čas izvajanja večji največ za faktor dve. Temu prispeva tudi zelo dobro prilagojen način izvajanja odstranjevanja smeti.

S prenosom knjižnica MPFR na Haskell, smo tako pridobili hitro programsko implementacijo števil v plavajoči vejici s poljubno fiksno natančnostjo v čistem funkcijskem

jeziku. Z njo omogočamo nadaljne raziskave v Haskell-u, ne da bi se bilo pri le-teh raziskovalcu potrebno ukvarjati s podrobnostmi, ki se dogajajo v ozadju.