Sustav za simboličko računanje u Haskellu

Čanadi, Vitomir

Master's thesis / Diplomski rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:899874

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-03**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Vitomir Čanadi

SUSTAV ZA SIMBOLIČKO RAČUNANJE U HASKELLU

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, srpanj, 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Uvod u Haskell **3**
1.1 Instalacija 4
1.2 Haskell primjeri 5

2 Osnovne funkcionalnosti **9**
2.1 Tipovi podataka 9
2.2 Funkcije 11

3 Normalizacija **13**
3.1 Dokaz stajanja funkcije *normalize* 13

4 Dodatne funkcionalnosti **23**
4.1 Sustavi linearnih jednadžbi 23
4.2 Testiranje QuickCheck-om 29

Bibliografija **33**

Uvod

Cilj ovog diplomskog rada je izrada sustava za simboličko računanje (*eng.* CAS - Computer algebra system) u Haskellu. Sustav za simboličko računanje je program u kojem korisnik može manipulirati matematičkim izrazima (termima). Takav sustav dopušta korisniku da konstruira izraze poput proizvoljne matematičke relacije ($e^{\pi i} + 1 = 0$), funkcije ($ax^2 + bx + c$) ili sustava linearnih jednadžbi. Uobičajeno se programerski matematička funkcija definira kao „crna kutija" koja za neki broj kao ulaznu vrijednost vraća drugi broj. Sama izgradnja izlazne vrijednosti je nevidljiva unutar tijela funkcije i korisnik koji koristi tu funkciju nema nikakav uvid u prirodu te funkcije, već samo može koristiti njezinu izlaznu vrijednost za danu ulaznu vrijednost. Za razliku od takvih funkcija, simboličke funkcije (izrazi) čuvaju dodatne informacije o samoj konstrukciji izraza. Nad takvim izrazima se onda mogu definirati operacije poput deriviranja, integriranja, računanja nultočke kvadratne funkcije ili rješavanja linearnog sustava, onako kako bi to čovjek napravio, manipuliranjem izraza na papiru.

Prvo poglavlje ovog diplomskog rada se sastoji od uvoda u programski jezik Haskell i od nekoliko primjera koji prikazuju razlike između Haskella i imperativnih programskih jezika. U drugom poglavlju opisane su glavne komponente programa poput definicija tipova podataka i funkcija koje barataju njima. Osnovna funkcionalnost CAS-a je normalizacija, odnosno pojednostavljivanje izraza. U trećem poglavlju dokazuju se svojstva funkcija za normalizaciju. Dodatna funkcionalnost je rješavanje linearnih sustava. Nadalje, u četvrtom poglavlju opisuju se funkcionalnosti, poput supstitucije i detekcije linearnosti izraza, koje su potrebne za rješavanje linearnih sustava. Na kraju četvrtog poglavlja opisana je funkcionalnost Haskellove biblioteke QuickCheck za testiranje svojstava programa. Konkretno, iskoristit ćemo QuickCheck za testiranje svojstava koja smo dokazali u trećem poglavlju.

Poglavlje 1

Uvod u Haskell

Haskell je funkcijski programski jezik. Razlika između imperativne i funkcijske paradigme je u tome da je program napisan u imperativnom jeziku konstruiran kao niz sekvencijalnih naredbi koje modificiraju varijable iz okoline (paradigma osnovana na RAM odnosno Turingovom stroju), dok je program napisan u funkcijskom jeziku opisan deklarativno kao skup izraza (paradigma osnovana na λ-računu). Konkretno, Haskell je baziran na sustavu F (*eng.* System F, Girard-Reynolds polymorphic λ-calculus, second-order λ-calculus). Sustav F je proširenje tipiziranog λ-računa kvantifikatorima nad tipovima (polimorfizam).

Najkorišteniji jezični prevoditelj za Haskell je *Glasgow Haskell Compiler* (skraćeno GHC). GHC omogućuje pisanje i testiranje Haskell koda te podržava brojne ekstenzije i biblioteke za generiranje izvršne datoteke. GHCi je sučelje za GHC. Unutar GHCi-a možemo interaktivno evaluirati Haskell izraze i testirati funkcije uvezene iz neke vanjske biblioteke ili iz vlastitog izvornog koda. Prednost GHCi-a je u tome da za testiranje određene funkcije programa ne moramo pisati glavni program te generirati izvršnu datoteku. GHCi omogućuje interaktivno učitavanje izvornog koda prilikom promjene, te testiranje konkretne funkcije iz koda, što rezultira povećanjem brzine razvoja programa.

Uz GHCi, koji je osnovni alat za rad u Haskellu, koristan alat je *Stack*. Prilikom razvoja nekog ozbiljnijeg projekta postaje jasna potreba za mnoštvom dodatnih vanjskih biblioteka. Svaka biblioteka može biti zavisna o drugačijoj verziji neke druge biblioteke. Već kod manjeg broja vanjskih biblioteka, postaje teško imati kontrolu nad međusobnim zavisnostima različitih biblioteka. Stack je napredniji alat za izgradnju Haskell projekata i upravljanje zavisnostima s vanjskim bibliotekama.

Hackage je javni repozitorij Haskell biblioteka. Stackage je skup kolekcija biblioteka određenih verzija čija međusobna kompatibilnost je testirana. Pomoću Stack-a možemo specificirati iz koje Stackage kolekcije želimo koristiti biblioteke, ovisno o potrebi. Cilj je uvijek koristiti najažurniju kolekciju, no ukoliko biblioteka koja nam je potrebna nema ažurnu verziju, tada moramo koristiti zadnju kolekciju biblioteka u kojoj je tražena bibli-

oteka kompatibilna s ostalima.

1.1 Instalacija

Upute za preuzimanje i instalaciju Stack-a mogu se naći ovdje:
`https://docs.haskellstack.org/en/stable/install_and_upgrade/`
Konkretno za operacijski sustav Windows:
`https://docs.haskellstack.org/en/stable/install_and_upgrade/#windows`

Ime Stack projekta ovog diplomskog rada je *cas*. Nakon instalacije Stack-a, potrebno je unutar konzole ući u direktorij projekta *cas* i pozvati naredbu *stack setup*. Ta naredba instalira GHC prevoditelj za verziju specificiranu unutar stack.yaml datoteke. Sada možemo pokrenuti interaktivno sučelje s naredbom *stack ghci*. Ukoliko je svaki prethodni korak bio uspješan, unutar ghci-a možemo testirati neke funkcionalnosti.
Na primjer: printSolveForWithBSDbg x012 linSys1

```
printSolveForWithBSDbg [_x0,_x1,_x2] linSys1
linSys:
-1 + x1 + 2*x0 + 3*x2 = 0
-3 + 2*x0 + 6*x1 + 8*x2 = 0
-5 + 6*x0 + 8*x1 + 18*x2 = 0
Finding Equ to solve for _x0...
Solving for _x0, Equ: -1 + x1 + 2*x0 + 3*x2 = 0
newRule:
_x0 --> 1/2 + -1/2*x1 + -3/2*x2
linSys:
-2 + 5*x1 + 5*x2 = 0
-2 + 5*x1 + 9*x2 = 0
Finding Equ to solve for _x1...
Solving for _x1, Equ: -2 + 5*x1 + 5*x2 = 0
newRule:
_x1 --> 2/5 + -1*x2
linSys:
4*x2 = 0
Finding Equ to solve for _x2...
Solving for _x2, Equ: 4*x2 = 0
newRule:
_x2 --> 0
linSysSolution:
```

```
rules:
_x0 --> 3/10
_x1 --> 2/5
_x2 --> 0
```

1.2 Haskell primjeri

Kako ne bismo odstupali od teme rada, za uvodne Haskell primjere uzet ćemo dijelove samog koda diplomskog rada. Primjeri koje prezentiramo su dobri za demonstraciju nekih svojstava Haskella, poput: funkcija višeg reda, parcijalne aplikacije, lijene evaluacije, η-redukcije i polimorfnih tipova.
No za početak, upoznajemo se sa sintaksom Haskella na jednom poznatijem primjeru.

Primjer 1.2.1.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

U prvom redu se definira tip funkcije *fac* kao funkcija koja uzima prirodan broj (Int) i vraća Int.
U drugom redu definiramo rezultat funkcije za ulaznu vrijednost 0.
Ukoliko ulazna vrijednost nije 0, rezultat funkcije je definiran u trećem redu kao produkt ulazne vrijednosti i povratne vrijednosti rekurzivnog poziva funkcije *fac*.

Primjer 1.2.2. Promatramo funkcije *normalizeIter* i *endoIter*.

```
normalizeIter :: Int -> T -> T
normalizeIter n = endoIter n normalizeStep
```

U prvom redu primjera definira se tip funkcije *normalizeIter*.
normalizeIter :: Int $\rightarrow T \rightarrow T$
normalizeIter je funkcija dvije varijable tipova Int i T (tip koji ćemo definirati u drugom poglavlju, predstavlja term CAS-a), te vraća tip T.
Funkcija *normalizeIter* za dani broj iteracija n i neki term t, vraća n-tu iteraciju funkcije normalizeStep na termu t. Takvu definiciju možemo opisati na sljedeći način: funkcija *normalizeIter*, za dani broj iteracija, vraća funkciju koja je dobivena komponiranjem funkcije *normalizeStep*, n puta.

Primijetimo da je tip funkcije *normalizeIter* jednak $Int \rightarrow T \rightarrow T$. Zbog desne asocijativnosti tipovskog operatora $(\rightarrow)$, to je ekvivalentno sa: $Int \rightarrow (T \rightarrow T)$. To jest, taj tip

je funkcija jedne *Int* varijable, koja vraća drugu funkciju tipa $T \rightarrow T$. Primijetimo da je u drugom redu koda, funkcija *normalizeIter* definirana samo za jedan parametar (n). Zato je izraz s desne strane jednakosti funkcija tipa $T \rightarrow T$, a to odgovara gornjoj definiciji tipa funkcije *normalizeIter*. Definicija funkcije na taj način zove se η-redukcija.

```
endoIter :: Int -> (a -> a) -> (a -> a)
endoIter n f = foldr (.) id (replicate n f)
```

Funkcija *normalizeIter* je definirana pomoću općenitije funkcije *endoIter*. Primijetimo da funkcija *endoIter* ima tip *Int* $\rightarrow$ *(a* $\rightarrow$ *a)* $\rightarrow$ *(a* $\rightarrow$ *a)*. No tip *a* nije tip poput Int ili T, *a* je tipovska varijabla. *endoIter* je funkcija dvije varijable. Prva varijabla je broj iteracija tipa Int, a druga je funkcija koja uzima bilo koji tip *a* i vraća taj isti tip. Funkcija *endoIter* kao rezultat vraća funkciju istog tipa kao funkcija koja joj je bila drugi parametar. Funkcije koje u svojem tipu sadrže tipovske varijable zovu se *polimorfne funkcije*.

Promotrimo definiciju funkcije *endoIter*.
Funkcija *replicate* je polimorfna funkcija koja uzima broj *n* tipa Int i *x* proizvoljnog tipa *a* te vraća *n* kopija elementa *x* u listi.
Primjeri:
replicate 3 "a" = ["a", "a", "a"]
replicate 5 12 = [12, 12, 12, 12, 12]
U slučaju funkcija *endoIter* element koji repliciramo je funkcija dana kao parametar (*f :: a* $\rightarrow$ *a*). Takvu listu funkcija nije moguće prikazati kao na primjer listu slova ili brojeva, no nad njome svakako možemo raditi određene operacije.
Na generiranu listu primjenjujemo polimorfnu funkciju *foldr*. Tip funkcije *foldr* je *(a* $\rightarrow$ *b* $\rightarrow$ *b)* $\rightarrow$ *b* $\rightarrow$ [*a*] $\rightarrow$ *b*. Unatoč naizgled kompliciranom tipu funkcije *foldr*, na primjeru se vidi da je njezina definicija jednostavna. Funkcija *foldr* kao prvi parametar uzima binarnu funkciju tipa *(a* $\rightarrow$ *b* $\rightarrow$ *b)* s kojom djeluje po nekoj listi. Drugi parametar je početna vrijednost tipa *b*. Treći parametar je [*a*] (lista *a*-ova). Rezultat je tipa *b*.
Primjeri:
foldr (+) 0 [1,2,3] = 6
foldr max 0 [2,3,1] = 3
foldr (.) id [\x $\rightarrow$ *x + 1, \x* $\rightarrow$ *3*x] = \x* $\rightarrow$ *3*x + 1*
Funkcija *(.)* je kompozicija dvije funkcije (u infiksnom obliku: $f \,.\, g$), *id* je polimorfna funkcija identiteta.

U definiciji funkcije *endoIter*, funkciju *foldr* koristimo kao u trećem primjeru. Binarna funkcija kojom djelujemo na listu funkcija je kompozicija. Početni element s kojim komponiramo je identiteta.
Primjer:

endoIter 4 *f*

$$= foldr\ (.)\ id\ (replicate\ 4\ f)$$
$$= foldr\ (.)\ id\ [f,\ f,\ f,\ f]\ =\ f\ .\ f\ .\ f\ .\ f\ .\ id$$

Cilj zadnjeg primjera bio je pokazati prednosti tretiranja funkcija kao običnih vrijednosti. Na taj način mogli smo sagraditi listu funkcija i djelovati na funkcije drugim funkcijama. Takve funkcije zovu se *funkcije višeg reda*.

Zadnji primjer demonstrira lijenu evaluaciju Haskella. Vrijednost *replicate 5 "a"* možemo izraziti i na sljedeći način:

$$take\ 5\ (repeat\ "a")$$

Funkcija *repeat* konstruira beskonačnu listu kopija elementa "a". Ukoliko pokušamo u GHCi-u unijeti izraz *repeat "a"*, generiranje rezultata neće nikada stati, no Haskell ne stvara tu vrijednost već pamti sam izraz *repeat "a"* kao beskonačnu listu elemenata "a", i na takvoj listi može dalje raditi operacije poput *take 5*. Funkcija *take* uzima broj *n* i neku listu i vraća prvih *n* elemenata te liste.

Definicija tipova u Haskell-u

U sljedećem primjeru opisujemo način definiranja tipova u Haskellu.

Primjer 1.2.3. Binarno stablo

```
data TreeA = LeafA
           | BranchA TreeA TreeA
```

Varijable tipa Tree mogu poprimiti dva tipa vrijednosti: *LeafA* i *BranchA l r*, gdje su *l* i *r* vrijednosti tipa *TreeA*.
Primjeri:

```
tA0 = LeafA
tA1 = BranchA LeafA (BranchA (BranchA LeafA LeafA) LeafA)
```

Ako želimo za dano stablo dohvatiti lijevo ili desno dijete potrebno je definirati posebne funkcije

```
childLA :: TreeA -> TreeA
childLA (BranchA l _) = l

childRA :: TreeA -> TreeA
childRA (BranchA _ r) = r
```

Kod složenijih tipova povećava se broj takvih funkcija koje treba posebno implementirati, no njihova definicija je jasna iz samog opisa tipa. U tu svrhu koristi se sljedeći način definiranja tipova.

```
data TreeB = LeafB
           | BranchB { childLB :: TreeB, childRB :: TreeB }
```

Iz definicije tipa, Haskell može zaključiti neke prirodne funkcionalnosti poput relacija *(==)*, *(<)* ili kako definirani tip prevesti u String (funkcija *show*). Bilo koju od tih relacija možemo sami implementirati za naš tip, ali ako želimo da Haskell to učini, to se postiže na sljedeći način:

```
data TreeB = LeafB
           | BranchB { childLB :: TreeB, childRB :: TreeB }
           deriving (Eq, Ord, Show)
```

Poglavlje 2

Osnovne funkcionalnosti

2.1 Tipovi podataka

Osnovna struktura nad kojom kod barata je term (u Haskellu označen s T).
Svaki term može biti jedno od sljedećeg:

- konstanta C: realni broj poput π ili e
- kompleksni broj F, gdje oznaka dolazi od engleske riječi za polje (field): $p + qi$ takvi da su $p, q \in \mathbb{Q}$. Na primjer 2/3 + 5i
- varijabla X: x0, x1, ...
- zbrajanje niza terma Add
- množenje niza terma Mul
- potenciranje terma termom Pow i prirodni logaritam Ln.

Neformalna definicija Terma

$C \rightarrow \pi \mid e$
$F \rightarrow 0 \mid 1 \mid -1 \mid \frac{1}{2} \mid \frac{-1}{2} \mid 1 + i$ ... (kompleksni brojevi s racionalnim koordinatama)
$X \rightarrow x_0 \mid x_1 \mid x_2 \mid ...$
$Add \rightarrow T + Add \mid T$
$Mul \rightarrow T * Mul \mid T$
$Pow \rightarrow T^T$
$Ln \rightarrow ln\ T$
$T \rightarrow C \mid F \mid X \mid Add \mid Mul \mid Pow \mid Ln$

Definicija terma u Haskellu

```
data C       = Pi   | E                    deriving (Eq, Ord, Show)
newtype X    = X    { xI     :: Int }     deriving (Eq,Ord)
newtype Add  = Add  { addTs  :: [T] }     deriving (Eq,Ord)
newtype Mul  = Mul  { mulTs  :: [T] }     deriving (Eq,Ord)

data T   = TF   { tF    :: F    }
         | TX   { tX    :: X    }
         | TC   { tC    :: C    }
         | TAdd { tAdd  :: Add  }
         | TMul { tMul  :: Mul  }
         | TPow { tPowB :: T
                , tPowE :: T    }
         | TLn  { tLnT  :: T    }
         deriving (Eq,Ord)
```

2.2 Funkcije

addCollapse, mulCollapse

Funkcije *addCollapse* i *mulCollapse* su jedne od bitnijih komponenta normalizacijskog koraka *collect*. Funkcija *addCollapse* za dani niz terma izgradi term tipa *Add*, te sve F-elemente unutar sume sumira u jedan F-element. Funkcija *mulCollapse* je definirana analogno za množenje.

```
addCollapse = add
        . filter (not . isZero)
        . ((:) <$> TF .  addF . filterF
              <*> filterNotF)


mulCollapse = any isZero ?>>> const zero
                         ||> mul
                           . filter (not . isOne)
                           . ((:) <$> TF . mulF . filterF
                                  <*> filterNotF)
```

Funkcije višeg reda

Primijetimo da su funkcije *addCollapse* i *mulCollapse* definirane s funkcijama višeg reda. Korisne funkcije višeg reda koje se koriste kao infix operatori su: <$>, <*>, ? ⋙ i ||>.
Za funkcije:

$$f :: b \to c \to d$$

$$g :: a \to b$$

$$h :: a \to c$$

vrijedi:

$$f <\$> g <*> h \;::\; a \to d$$

$$(f <\$> g <*> h)\ x = f\ (g\ x)\ (h\ x)$$

Za funkcije:

$$p :: a \to Bool$$

$$f :: a \to b$$

$$g :: a \to b$$

vrijedi:

$$p \;?\ggg\; f \;||>\; g :: a \to b$$

$$(p \;?\ggg\; f \;||>\; g)\ x = if\ (p\ x)\ then\ (f\ x)\ else\ (g\ x)$$

mulParts, powParts

Funkcije *mulParts* i *powParts* također se koriste prilikom normalizacijskog koraka *collect*. One odvajaju F izraze od ne-F izraza unutar sume (produkta).

```
mulParts :: T -> (F,T)
mulParts = isF     ?>>>  (,) <$> tF
                         <*> const one
       ||> isMul  ?>>> ((,) <$> mulF . filterF
                         <*> mul . filterNotF ) . tMulTs
       ||>              (,) <$> const oneF
                         <*> id

powParts :: T -> (T,T)
powParts = ((&&) <$> isPow
                 <*> isF . tPowE) ?>>> (,) <$> tPowE <*> tPowB
                                      ||> (,) <$> const one <*> id
```

tmap

Funkcija *tmap* je funkcija koja je analogna Haskellovoj funkciji *fmap*. Za danu transformaciju terma f i za term t, *(tmap f) t* je term dobiven primjenom funkcije f na svakom čvoru terma t.

```
tmap = tmapDepthCond (const True) (-1)

tmapToDepth, tmapOnDepth, tmapFromDepth :: Int -> (T -> T) -> T -> T
[tmapToDepth,tmapOnDepth, tmapFromDepth] =
    fmap tmapDepthCond [(>=0), (==0), (<=0)]

tmapDepthCond :: (Int -> Bool) -> Int -> (T -> T) -> T -> T
tmapDepthCond p d f =
    bool id f (p d)
  . (   isAdd ?>>> add . fmap rec . tAddTs
    ||> isMul ?>>> mul . fmap rec . tMulTs
    ||> isPow ?>>> (pow <$> rec . tPowB
                        <*> rec . tPowE )
    ||> isLn  ?>>> ln . rec . tLnT
    ||>            id
    )
    where
    rec = tmapDepthCond p (pred d) f
```

Poglavlje 3

Normalizacija

Jedna od osnovnih funkcionalnosti svakog sustava za simboličko računanje je pojednostavljivanje složenijih izraza u jednostavnije, što naš sustav čini funkcijom *normalize*. Pozivanje funkcije *normalize* je automatsko prilikom poziva nekog od operatora (+, *, **). Kako bi se osiguralo stajanje izvršavanja naredbi CAS-a, dokazujemo stajanje funkcije *normalize*. To se dokazuje uvođenjem pomoćne funkcije *norm*, koja je strogo padajuća sa svakim pozivom normalizacijskog koraka tako dugo dok se izraz (term) mijenja djelovanjem funkcije *normalizeStep*.

Napomena 3.0.1. *Oznake:*
[n] = {1..n}
S_n = { p : [n] → [n] | p je bijekcija}
x [t_1, .., t_n] *: čvor tipa* x *(Add,Mul,Pow,...) sa podčvorovima* t_1, .., t_n
t |>| r :⇔ norm t > norm r
t |>.| r :⇔ t = r ∨ t |>| r

3.1 Dokaz stajanja funkcije *normalize*

Funkcija **normalize** iterira funkciju **normalizeStep** tako dugo dok ona ne konvergira do fiksne točke. Finalni izraz se sortira funkcijom **sort**.

```
normalize = sort . normalize'
    where
    normalize' t
        | isFixT normalizeStep t      = t
        | nonDecNorm normalizeStep t = error $ "non-monotone norm"
        | otherwise                   = normalize' (normalizeStep t)
```

normalizeStep je kompozicija funkcija **flatten** i **collect**.

```
normalizeStep = collect . flatten
```

Funkcija *flatten* (*collect*, *sort*) je definirana kao **tmap** po termu s funkcijom *flattenNode* (*collectNode*, *sortNode*). *tmap f* iterira funkciju *f* po stablu terma (od listova prema gore) tako da se funkcija *flattenNode (collectNode, sortNode)* definira samo za čvor stabla.

```
flatten = tmap flattenNode
collect = tmap collectNode
sort    = tmap sortNode
```

(Pregled funkcija tmap, flattenNode, collectNode, sortNode, kasnije u dokazima.)

Cilj je dokazati da funkcija *normalize* staje. U tu svrhu konstruira se tip **Norm** i funkcije **norm** , **decNorm**, **nonDecNorm**, **isFixT** , **isNormalFor**, **isCumulativeFor** (koristimo je kao infiksni operator *f `isNormalFor` t == isNormalFor f t*) , **isQuasiDecFor**, **isFixTIndex**, **nonDecNormIndex** , **converges**. Također se izvan koda definiraju dva svojstva (**isNormal**, **isCumulative** i **isQuasiDec**) koja nisu definirana u samom kodu zbog nemogućeg opisa kvantificiranja $\forall t \in T$ (*f* ‘*isNormalFor*‘ *t*) u kodu. Takvo svojstvo bi trebalo provjeravati beskonačno terma kako bi vratilo rezultat.
Norm je tip s leksikografskim uređajem i operacijom zbrajanja po komponentama. To je tip koji vraća funkcija norm i njegov strogi pad prilikom iteracija *normalizeStep* se promatra u dokazu. Ključno je da je uređaj na *Norm* dobar, odnosno *Norm* je izomorfan ordinalu ω^3.

Sljedeće funkcije govore o odnosu *norm t* i *norm (f t)* za neke *f* i *t*. Funkcija *decNorm* (*decreasing Norm*) vraća True, ako djelovanje *f* na *t* strogo smanjuje normu. Funkcija *isFixT*, vraća True ako *f* ne mijenja *t*. Funkcija *isNormalFor* govori o „dobrom ponašanju” djelovanja *f* na *t*, što znači da *f* ili ne izmijeni *t* ili mu strogo smanji normu. Funkcija *isNormal* je samo konjunkcija primjene funkcije *isNormalFor* nad svim termima (definicija niže). Funkcije *isCumulative* i *isQuasiDec* su jače varijante funkcije *isNormalize* koje su potrebne prilikom dokaza propozicije 3.1.7.

```
newtype Norm = Norm (Int,Int,Int) deriving (Eq,Ord,Show)
instance Num Norm where
    (Norm (x,y,z)) + (Norm (w,v,q)) = Norm (x+w,y+v,z+q)
    fromInteger i = Norm (fromIntegral i,fromIntegral i,fromIntegral i)
norm :: T -> Norm
norm t = Norm ( lengthNonFLeafs t
              , lengthFs t
              , lengthNonFlats t)

infix 4 |>.|, |>|
(|>.|),(|>|) :: T -> T -> Bool
decNorm, nonDecNorm, isFixT, isNormalFor, isCumulativeFor :: (T -> T)
   -> T -> Bool
isQuasiDecFor :: (T -> T) -> T -> T -> Bool

t |>| r = norm t > norm r
t |>.| r = sort t == sort r || t |>| r
decNorm f t = t |>| f t
nonDecNorm f = not . decNorm f
isFixT f t = sort t == sort (f t)
```

Definicija 3.1.1. *Uvodimo oznake:*

$$isNormal\ f :\Leftrightarrow \forall t \in T\ (f\ `isNormalFor`\ t)$$

$$isCumulative\ f :\Leftrightarrow \forall t \in T\ (f\ `isCumulativeFor`\ t)$$

$$isQuasiDec\ f :\Leftrightarrow \forall t, r \in T\ (isQuasiDecFor\ f\ t\ r)$$

Definicije funkcija *isNormalFor*, *isCumulativeFor*, *isQuasiDecFor* u Haskellu:

```
f `isNormalFor` t = isFixT f t || decNorm f t
```

```
f `isCumulativeFor` t =
       isLeaf t
    || sum (fmap norm $ selectAllChilds t) >= norm (f t)
```

```
isQuasiDecFor f t r =
                 eqType t r
            &&  (not . isLeaf) t
            &&  tLengthAllChilds t == tLengthAllChilds r
            &&  (((and .) . zipWith (|>.|)) `on` selectAllChilds)  t r
      `implies` t |>.| f r
```

S obzirom da su funkcije *isCumulativeFor* i *isQuasiDecFor* složenije ovo je jasniji opis funkcija *isCumulative* i *isQuasiDec* s raspisanim *isCumulativeFor* i *isQuasiDecFor* funkcijama:

$$isCumulative\ f = \forall t{=}x\ [t_1, ..t_n] \in T\ (\sum_{i=1}^{n}(norm\ t_i) \geq norm\ (f\ t))$$

$$isQuasiDec\ f = \forall\ t{=}x\ [t_0, .., t_n],\ r{=}x\ [r_0, .., r_n],\ i \in [n]\ (t_i\ |{>}.|\ r_i\ \rightarrow\ t\ |{>}.|\ f\ r)$$

Funkcija *fixTIndex* (*nonDecNormIndex*) vraća prvu iteraciju i za koju vrijedi f^i t == f^{i+1} (not $ norm (f^i t) > norm (f^{i+1} t)). Funkcija *converges f t* ispituje je li norma padajuća funkcija po indeksu iteracije funkcije f nad početnim termom t na domeni od nulte iteracije do iteracije fiksne točke.

```
fixTIndex, nonDecNormIndex :: (T -> T) -> T -> Int
fixTIndex f = fromMaybe (error "no fixT")
            . findIndex (isFixT f) . iterate f

nonDecNormIndex f = fromMaybe (error "no nonDecNorm")
                  . findIndex (nonDecNorm f) . iterate f

converges :: (T -> T) -> T -> Bool
converges f = (==) <$> fixTIndex f <*> nonDecNormIndex f
```

Teorem 3.1.2. *Za svaki term t vrijedi:*

$$converges\ normalizationStep\ t\ .$$

Dokaz. Iz definicije funkcije *converges* vidi se da se gornji teorem dokazuje tako da se dokaže da funkcija normalizeStep strogo smanji normu proizvoljnog terma *t*, dakle da vrijedi *normalizeStep 'isNormalFor' t*. S obzirom da je *norm t i :: Norm*, te je tip *Norm* dobro uređen, gornji uvjet osigurava stajanje funkcije *normalize*. Prethodni korak (*normalizeStep 'isNormalFor' t*) se dokazuje tako da se dokaže *flatten 'isNormalFor' t* i *collect 'isNormalFor' t*. Za to je potrebno dokazati da vrijedi *isQuasiDec flattenNode* i *isQuasiDec collectNode*, te *isQuasiDec f* → *isNormal (tmap f)* . □

Napomena 3.1.3. *S obzirom da funkcije* flattenNode*,* sortNode *i* collectNode *ne mijenjaju terme tipova C, F, X i Ln, promatramo samo djelovanja tih funkcija na terme tipa Add i Mul.*

Propozicija 3.1.4. *Vrijedi:*

$$isCumulative flattenNode \wedge isNormal flattenNode$$

Dokaz.
Definicija funkcije *flattenNode*:

```
flattenNode = isAdd ?>>> add . concatMap addElems . tAddTs
            ||> isMul ?>>> mul . concatMap mulElems . tMulTs
                       ||> id
```

Dokazujemo za čvor Add. Dokaz za čvor Mul je analogan jer *flattenNode* ne ulazi u neka interna svojstva Add i Mul čvorova već ih promatra jednako, kao čvorove stabla s oznakom i podčvorovima.
Dakle, $t = t_1 + .. + t_n$. Neka su $k \in [n]$ i $p \in S_n$ takvi da je tip od t_{p_i} jednak Add upravo za prvih k podizraza, te vrijedi: $t_{p_i} = t_{(p_i,1)} + .. + t_{(p_i,k_{p_i})}$.
Tada vrijedi: $flattenNode\ t = ((t_{(p_1,1)} +..+ t_{(p_1,k_{p_1})}) +..+ (t_{(p_k,1)} +..+ t_{(p_k,k_{p_k})})) + t_{p_{k+1}} +..+ t_{p_n}$.
Ukoliko ne postoji ni jedan Add podčvor, izraz se ne mijenja i vrijedi tvrdnja propozicije, stoga pretpostavimo da postoji barem jedan Add podčvor. Zbog postojanja Add podčvora, *norm t* ima bar za 1 veću treću koordinatu. Nakon primjene funkcije *flattenNode*, taj čvor pridodaje 0 trećoj koordinati od *norm (flattenNode t)*. Iz razloga što se podčvorovi tipa Add uklanjaju te se njihovi podčvorovi dodaju u listu podčvorova od t, flattenNode ne mijenja listove, tako da prva i druga koordinata *Norm-a* ostaju iste. Vrijedi:

$$\sum_{i=1}^{n} (norm\ t_i) \geq norm\ (flattenNode\ (x\ [t_1, .., t_n]))$$

to jest *isCumulative flattenNode*.
Također iz prethodnog vrijedi:

$$t \mid>. \mid flattenNode\ t$$

to jest *isNormal flattenNode* □

Propozicija 3.1.5. *Vrijedi:*

$$isCumulative\ collectNode \wedge isNormal\ collectNode$$

Dokaz.
Definicija funkcije *collectNode*:

```
collectNode TAdd {..}    = collectAddNode tAdd
collectNode TMul {..}    = collectMulNode tMul
collectNode t@TPow {..} = collectMulNode (Mul [t])
collectNode t             = t

collectAddNode :: Add -> T
collectAddNode =
     addCollapse
   . fmap (uncurry mulBinCollapse)
   . fmap (bimap TF id)
   . fmap (foldlFst addFBin zeroF)
   . stableGroupOn snd
   . fmap mulParts
   . addTs

collectMulNode :: Mul -> T
collectMulNode =
     mulCollapse
   . fmap (uncurry $ flip powCollapse)
   . fmap (foldlFst addBinCollapse zero)
   . stableGroupOn snd
   . fmap powParts
   . mulTs
```

Slučaj 1) t je tipa Add:
Neka je $t = t_1 + .. + t_n$. *collectNode* prvo zapisuje t u obliku $a_1 * u_1 + .. + a_n * u_n$ gdje su a_i tipa F, a u_i tipa T (ako t_i nema F-faktor, onda je $a_i = 1$). Neka su $0 = x_1 < x_2 < .. < x_k < n$ indeksi i $p \in S_n$ takvi da vrijedi:

$$u_{p_{x_1+1}} = u_{p_{x_1+2}} = \cdots = u_{p_{x_2}}$$

$$u_{p_{x_2+1}} = u_{p_{x_2+2}} = \cdots = u_{p_{x_3}}$$

$$\dots$$

$$u_{p_{x_k+1}} = u_{p_{x_k+2}} = \cdots = u_{p_n}$$

tada *collectNode t* izgleda ovako:

$$(a_{p_{x_1+1}} + \cdots + a_{p_{x_2}}) * u_{p_{x_1+1}} \ + (a_{p_{x_2+1}} + \cdots + a_{p_{x_3}}) * u_{p_{x_2+1}} \ + \cdots + (a_{p_{x_k+1}} + \cdots + a_{p_n}) * u_{p_{x_k+1}}$$

Svi F elementi se sumiraju u jedan F element, stoga *collectNode t* izgleda ovako:

$$b_1 * t_{p_{x_1+1}} \ + b_2 * t_{p_{x_2}+1} \ + \cdots + b_k * t_{p_{x_k}+1}$$

Također, ako je neki $t_{p_{x_i+1}}$ iz F, tada se on pomnoži s b_i u jednu vrijednost tipa F, pa se zato dalje promatraju t-ovi koji nisu uz F.
Ako je neki b_i jednak 1, tada se b_i ukloni iz produkta tako da se ne stvaraju dodatni F listovi, na primjer x0 = 1*x0 → x0. Jedini potencijalno dodani F listovi koji se stvaraju su uz neki element koji se pojavljuje više puta u sumi. Ako je taj element bio pomnožen F-listom, onda nema stvaranja novog F lista (na primjer 2*x0 + 3*x0 = 5*x0), no ako element prvobitno nije bio pomnožen F-listom (na primjer x0 + x0 = 2*x0) tada se dodaje množenje elementom iz F. Ipak, to se može dogoditi kada smo izlučili neki element koji se ponavlja u sumi više od jednom, i u tom slučaju nestaje bar jedna pojava tog elementa (x0). Taj element je podizraz i u svojem stablu ima (F ili ne-F) listove tako da se njihov broj smanji, pa se uvijek smanjuje broj ne-F čvorova, ili broj ne-F čvorova ostaje isti, a broj F čvorova se smanji.

Slučaj za t je tipa Mul dokazuje se analogno. Jedina razlika je da je operacija zbrajanja zamijenjena množenjem, a operacija množenja, potenciranjem.

Vrijedi:

$$\sum_{i=1}^{n} (norm\ t_i) \geq norm\ (collectNode\ (x\ [t_1, .., t_n]))$$

to jest *isCumulative collectNode*.
Iz prethodnog također vrijedi:

$$t\ |>.|\ collectNode\ t$$

to jest *isNormal collectNode* □

Propozicija 3.1.6. *Za svaku transformaciju izraza f, vrijedi:*

$$isCumulative\ f \wedge isNormal\ f \rightarrow isQuasiDec\ f$$

Dokaz. Neka je f funkcija takva da vrijedi *isCumulative f* to jest

$$\forall t = x\ [t_1, ..t_n] \in T\ \left(\sum_{i=1}^{n}(norm\ t_i) \geq norm\ (f\ t)\right)$$

i *isNormal f* to jest

$$\forall t\ (t\ \ |>.|\ f\ t)$$

Kako bismo dokazali da vrijedi isQuasiDec f potrebno je dokazati da za svaka dva terma t i r vrijedi:
isQuasiDecFor f t r odnosno

$$\forall i \in [n]\ (t_i\ \ |>.|\ \ r_i) \rightarrow t\ \ |>.|\ \ f\ r$$

Pretpostavke teorema:

$$t = x\ [t_1, .., t_n]$$

$$r = x\ [r_1, .., r_n]$$

$$\forall i \in [n]\ (t_i\ \ |>.|\ \ r_i)$$

Cilj je dokazati da vrijedi: $t\ \ |>.|\ f\ r$
Ako vrijedi $\forall i \in [n]\ (t_i = r_i)$ tada je $t = r$ pa iz *isNormal f* slijedi $t\ \ |>.|\ f\ r$, odnosno *isQuasiDec f*
Ako postoji neki i takav da vrijedi $t_i\ |>|\ r_i$, tada vrijedi:

$$norm\ t$$

(po definiciji funkcije norm)

$$\geq \sum_{i=1}^{n}(norm\ t_i)$$

(pretpostavka teorema + postojanje i takavog da vrijedi $t_i\ |>|\ r_j$)

$$> \sum_{i=1}^{n}(norm\ r_i)$$

(zbog *isCumulative f*)

$$\geq norm\ (f\ r)$$

Vrijedi *isQuasiDec f.* □

Propozicija 3.1.7. *Za svaku transformaciju izraza f, vrijedi:*

$$isQuasiDec\ f \rightarrow isNormal\ f$$

Dokaz. Direktno iz definicije *isQuasiDec* za t=r. □

Propozicija 3.1.8. *Za svaku transformaciju izraza f, vrijedi:*

$$isQuasiDec\ f \rightarrow isNormal\ (tmap\ f)$$

Dokaz.
Neka je f neka funkcija za koju vrijedi *isQuasiDec f*.
Cilj je dokazati da za svaki term t vrijedi *(tmap f) 'isNormalFor' t*. Dokaz provodimo indukcijom po stablu tipa T.

- List: Po definiciji funkcije *tmap*, ako je t list tada vrijedi:
  (tmap f) t = f t.
  Po pretpostavci teorema vrijedi *isQuasiDec f*, zamjenom f sa *tmap f* u definiciji svojstva *isQuasiDec*, vrijedi: *isQuasiDec (tmap f)* za listove. Po propoziciji 3.1.7 vrijedi: *isNormal (tmap0 f)* za listove.

- Ne-list: Pretpostavimo da za neke $t_i \in T, i \in [n]$ vrijedi:
  (tmap f) 'isNormalFor' t_i
  odnosno t_i |>.| $(tmap\ f)\ t_i$
  Za term t = x[$t_1, .., t_n$], vrijedi:

  $$(tmap\ f)\ x[t_1, .., t_n]$$

  (po definiciji funkcije *tmap*)

  $$= f\ x[(tmap\ f)\ t_1, .., (tmap\ f)\ t_n]$$

  (zbog *isQuasiDec f* i pretpostavke indukcije)

  $$|.<|\ x[t_1, .., t_n] = t$$

Dakle, vrijedi:

$$t\ |>.|\ (tmap\ f)\ t$$

odnosno *isNormal (tmap f)*. □

Propozicija 3.1.9. *Vrijedi:*

$$isNormalflatten \wedge isNormalcollect \wedge isNormalnormalizeStep$$

Dokaz. Iz 3.1.4 vrijedi:

$$isCumulative\ flattenNode\ \wedge\ isNormal\ flattenNode$$

Iz 3.1.6 vrijedi:

$$\forall f\ (isCumulative\ f\ \wedge\ isNormal\ f -> isQuasiDec\ f)$$

$$\Rightarrow isQuasiDec flattenNode$$

Iz 3.1.8 vrijedi:

$$\forall f\ (isQuasiDec\ f -> isNormal\ (tmap\ f))$$

$$\Rightarrow isNormal\ (tmap\ flattenNode)$$

$$\Leftrightarrow isNormal\ flatten$$

Analogno za collect.
Zbog *normalizeStep* = *collect* . *flatten* vrijedi:

$$norm\ t > norm\ (flatten\ t) > norm\ (collect\ (flatten\ t)) = norm\ (normalizeStep\ t)$$

odnosno *isNormal normalizeStep*. □

Poglavlje 4

Dodatne funkcionalnosti

4.1 Sustavi linearnih jednadžbi

Definicije tipova i funkcija

Osim osnovnih uvjeta za CAS poput normalizacije, jedan od najosnovnijih funkcijskih uvjeta je mogućnost rješavanja sustava linearnih jednadžbi. Za razliku od standardnih numeričkih metoda za rješavanje sustava preko matrica Gaussovim eliminacijama, u simboličkom računanju se oslanjamo na funkcionalnost manipuliranja izrazima, i rješavamo sustav izražavanjem pojedine varijable u svakoj jednadžbi te uvrštavanjem u ostale jednadžbe. U tu svrhu definirani su sljedeći tipovi podataka:

- *Equ* (Equation)
  Sadrži term equT koji predstavlja jednadžbu *equT = 0*.

```
data Equ    = Equ    { equT          :: T      }  deriving (Eq,Ord)
```

- *LinSys*
  Sadrži listu jednadžbi *linSysEqus*.

```
data LinSys = LinSys { linSysEqus :: [Equ] }  deriving (Eq,Ord)
```

- *Rule*
  Pravilo koje opisuje supstituciju varijable termom. Sadrži varijablu *ruleX* (tip X) i term *ruleT*. U kodu se konstrukcija pravila opisuje s pomoćnom funkcijom *(-->)*. Na primjer *x0 --> 3*x1+1*.

```
data Rule   = Rule {ruleX::X, ruleT::T}
(-->) :: X -> T -> Rule
(-->) = Rule
```

- *LinSysSolution*
  Predstavlja rješenje sustava u obliku liste supstitucijskih pravila (*solutionRules*) za svaku varijablu sustava, te liste orgraničenja (*constraints*) ukoliko sustav nema rješenja ili ima više rješenja.

```
data LinSysSolution = LinSysSolution { solutionRules :: [Rule]
                                     , constraints   :: [Equ]  }
```

Osim tipova podataka, bitnije funkcije za rješavanje sustava su:

- *solveFor*
  Funkcija koja kao parametre uzima linearni sustav i skup varijabli, te vraća tip LinSysSolution.

```
solveFor :: [X] -> LinSys -> LinSysSolution
solveFor []     (LinSys equs) = emptySolutionWithConstr equs
solveFor xs     (LinSys [])   = emptySolution
solveFor (x:xs) linSys        =
    case findEquForX linSys x of
        Nothing  -> solveFor xs linSys
        Just equ ->
            let newRule           = solveForXEqu x equ
                linSysWithNewRule = substituteInLinSys
                                        newRule
                                        (removeEqu equ linSys)
            in  addRule newRule $
                        solveFor xs linSysWithNewRule
```

- *expressXFromT*
  Funkcija koja za danu varijablu i jednadžbu vraća pravilo koje izražava varijablu iz jednadžbi. Na primjer, *expressXFromT x0 (2*x0 + 3*x1 + 5 = 0) = x0 ⇢ -3/2*x1 - 5/2*.

```
expressXFromEqu :: X -> Equ -> T
expressXFromEqu x Equ {..} = expand $ expressXFromT x (normalize
    equT)
    where
    expressXFromT :: X -> T -> T
    expressXFromT x t@TAdd {..} =
        mul [ aInv $ membersWithoutX x tAdd
            , mInv $ removeXFromMembers x $  membersWithX x tAdd
            ]
    expressXFromT x t           = zero
```

- *substitute*
  Funkcija koja za dano pravilo $x \dashrightarrow t$ i term r vraća term dobiven zamjenom svake pojave varijable x termom t, unutar terma r.

```
substituteNode :: Rule -> T -> T
substituteNode (Rule x r) TX {..} | tX == x = r
substituteNode _           t                 = t

substitute :: Rule -> T -> T
substitute rule t = expand $ tmap (substituteNode rule) t
```

Primjeri sustava

Navodimo neke primjere koji se nalaze u Cas.hs datoteci. Poziv funkcije koja rješava sustav izgleda ovako, za primjer 1.: *solveForWithBS* [*_x0,_x1,_x2*] *linSys1*. Prvi parametar je tipa [*X*] to jest lista varijabli. Razlika između *x0* i *_x0* je u tome da je *x0* term TX (X 0), a *_x0* je varijabla X 0. *x012* je sinonim za [*_x0,_x1,_x2*]. Za računanje finalnog rezultata linearnog sustava koristi se funkcija *solveForWithBS*. Druga varijanta je funkcija *solveFor*, koja ne radi supstituciju unatrag. U sljedećim primjerima koristit ćemo varijantu *printSolveForWithBSDbg* (Debug), koja nije namijenjena za korištenje, osim u svrhe testiranja jer u svakom koraku ispisuje stanje linearnog sustava. U sljedećim primjerima promatramo sustave *linSys1, linSys2, linSys3, linSys4* zapisane u Haskell kodu, te rezultate djelovanja funkcije *printSolveForWithBSDbg* na njima.

- Primjer 1. (jedinstveno rješenje)

```
linSys1 =
        2*x0 +   x1 +  3*x2 === 1
   &&: 2*x0 + 6*x1 +  8*x2 === 3
   &&: 6*x0 + 8*x1 + 18*x2 === 5
```

  Ispis:

```
printSolveForWithBSDbg [_x0,_x1,_x2] linSys1
linSys:
-1 + x1 + 2*x0 + 3*x2 = 0
-3 + 2*x0 + 6*x1 + 8*x2 = 0
-5 + 6*x0 + 8*x1 + 18*x2 = 0
Finding Equ to solve for _x0...
Solving for _x0, Equ: -1 + x1 + 2*x0 + 3*x2 = 0
newRule:
_x0 --> 1/2 + -1/2*x1 + -3/2*x2
```

```
linSys:
-2 + 5*x1 + 5*x2 = 0
-2 + 5*x1 + 9*x2 = 0
Finding Equ to solve for _x1...
Solving for _x1, Equ: -2 + 5*x1 + 5*x2 = 0
newRule:
_x1 --> 2/5 + -1*x2
linSys:
4*x2 = 0
Finding Equ to solve for _x2...
Solving for _x2, Equ: 4*x2 = 0
newRule:
_x2 --> 0
linSysSolution:
rules:
_x0 --> 3/10
_x1 --> 2/5
_x2 --> 0
```

- Primjer 2. (više rješenja)

```
linSys2 =
       3*x0 +       x1 + (-6)*x2 === -10
  &&: 2*x0 +       x1 + (-5)*x2 === -8
  &&: 6*x0 + (-3)*x1 +     3*x2 === 0
```

Ispis:

```
printSolveForWithBSDbg [_x0,_x1,_x2] linSys2
linSys:
10 + x1 + -6*x2 + 3*x0 = 0
8 + x1 + -5*x2 + 2*x0 = 0
-3*x1 + 3*x2 + 6*x0 = 0
Finding Equ to solve for _x0...
Solving for _x0, Equ: 10 + x1 + -6*x2 + 3*x0 = 0
newRule:
_x0 --> -10/3 + -1/3*x1 + 2*x2
linSys:
4/3 + 1/3*x1 + -1*x2 = 0
-20 + -5*x1 + 15*x2 = 0
```

```
Finding Equ to solve for _x1...
Solving for _x1, Equ: 4/3 + 1/3*x1 + -1*x2 = 0
newRule:
_x1 --> -4 + 3*x2
linSys:
0 = 0
Finding Equ to solve for _x2...
No Equ solvable for _x2
linSysSolution:
rules:
_x0 --> -2 + x2
_x1 --> -4 + 3*x2
```

- Primjer 3. (nema rješenja)

```
linSys3 =
       x0 +            x2 === 1
  &&: x0 +       x1 + x2 === 2
  &&: x0 + (-1)*x1 + x2 === 1
```

Ispis:

```
printSolveForWithBSDbg [_x0,_x1,_x2] linSys3
linSys:
-1 + x0 + x2 = 0
-2 + x0 + x1 + x2 = 0
-1 + x0 + x2 + -1*x1 = 0
Finding Equ to solve for _x0...
Solving for _x0, Equ: -1 + x0 + x2 = 0
newRule:
_x0 --> 1 + -1*x2
linSys:
-1 + x1 = 0
-1*x1 = 0
Finding Equ to solve for _x1...
Solving for _x1, Equ: -1 + x1 = 0
newRule:
_x1 --> 1
linSys:
-1 = 0
```

```
Finding Equ to solve for _x2...
No Equ solvable for _x2
linSysSolution:
no solution
```

- Primjer 4. (parametrizirano rješenje)

```
linSys4 =
          x0 + x3*x1 === 1
   &&:  2*x0 +  6*x1  === 3
```

Ispis:

```
printSolveForWithBSDbg [_x0,_x1] linSys4
linSys:
-1 + x0 + x1*x3 = 0
-3 + 2*x0 + 6*x1 = 0
Finding Equ to solve for _x0...
Solving for _x0, Equ: -1 + x0 + x1*x3 = 0
newRule:
_x0 --> 1 + -1*x1*x3
linSys:
-1 + -2*x1*x3 + 6*x1 = 0
Finding Equ to solve for _x1...
Solving for _x1, Equ: -1 + -2*x1*x3 + 6*x1 = 0
newRule:
_x1 --> (6 + -2*x3)**-1
linSysSolution:
rules:
_x0 --> 1 + -1*(6 + -2*x3)**-1*x3
_x1 --> (6 + -2*x3)**-1
```

4.2 Testiranje QuickCheck-om

U trećem poglavlju dokazivali smo svojstva koja funkcije za normalizaciju (*flattenNode*, *collectNode*, ...) moraju zadovoljiti kako bi algoritam normalizacije stao. Konkretno, za svaki term *t*, vrijednost *norm t* je opisivala neke karakteristike terma *t* (broj ne-F listova, broj F listova, broj čvorova s djetetom istog tipa). Pomoću tih karakteristika može se opisati željeno ponašanje funkcije za normalizaciju. Cilj nam je naći takve karakteristike čija vrijednost se stogo smanjuje primjenom normalizacijske funkcije. Problem je u tome da je teško definirati karakteristike a priori, zbog svih mogućih rubnih slučajeva koji predstavljaju kontraprimjer. Recimo, trenutna definicija tipa *Norm* ne opisuje „dobro" ovakvu normalizaciju $2 * 2^{x_0} \rightarrow 2^{x_0+1}$. Takav primjer i još mnogo drugih koji mogu postojati za različite odabire norme je teško uočiti. Motivacija za dokazivanje određenih svojstva je njihov prolaz na velikom broju testova. Pomoću Haskellove biblioteke QuickCheck možemo definirati takve testove. To se postiže tako da definiramo način izgradnje proizvoljnog terma. Ako za neki tip želimo da ima svojstvo generiranja proizvoljne vrijednosti, potrebno je implementirati instancu QuickCheck-ove klase *Arbitrary*. Samo izvršavanje testova postiže se pozivom funkcije *quickCheck*. Kao parametar joj prosljeđujemo funkciju koju želimo testirati (na primjer *isNormalFor*). Takva funkcija mora kao parametre imati tipove koji imaju *Arbitrary* instancu, te mora vraćati Bool vrijednost. QuickCheck automatski izvršava određeni broj testova generirajući proizvoljne ulazne vrijednosti, te provjerava da li funkcija vraća True.
Za razliku od funkcije *quickCheck*, funkcija *quickCheckWith* nam omogućuje definiranje ukupnog broja testova.
Primjer poziva funkcije *quickCheckWith*:
quickCheckWith stdArgs { maxSuccess = 1000} (isNormalFor normalizeStep)
Funcije *verboseCheck* i *verboseCheckWith* ispisuju svaki test.
Primjer ispisa izvršavanja 10 testova:

```
verboseCheckWith stdArgs {maxSuccess = 10} (isNormalFor normalizeStep)
Passed:
(-1+-1i)
Passed:
ln 0**ln x0
Passed:
ln ln x1
Passed:
(ln (Pi + Pi + x0) + Pi**1/2 + (-1+i)**(1/2+1/2i))*
ln (E + Pi + E + ln E)
*ln (x0**E)*(x0*(-1/2+-1/2i)*(1 + -1 + E)*1/2*Pi*x0*x1*E**x0)**
(ln x0 + 0**(1+1/2i))
```

```
Passed:
ln (Pi + Pi + (-1+i) + Pi)
Passed:
ln (E**ln E)
Passed:
ln (E**x1*ln (1/2+-1i)*(E + E + 1/2i + x1 + (1/2+-1/2i) + x0))
Passed:
ln (x1*-1*Pi*x1) + -1i**(x0*Pi*(1+1/2i)*x0*(1+1/2i)) + -1/2i
+ -1/2i + x1 + x1 +
(ln x0 + E*Pi)*(-1**(-1/2+i))**(x1 + Pi + x1 + E + E)
*ln -1i*E*x1*Pi*E*(-1+i)
Passed:
ln x1*ln ((-1+-1i)**x0)*(x0*(1/2+i)*Pi*E + x0 + x1 + Pi + x0)
+ 0**(-1/2**x0)*((-1+i) +
 -1/2i + Pi**i + x0 + x1) + ln (ln (x0*E*Pi)*ln Pi*x0*
 ln ln -1/2*E*E*(-1/2i + (-1/2+1/2i) + Pi)**x1*x0*x0*Pi)
Passed:
((E*(-1/2+-1i))**1/2)**(-1**(1/2+i))*(E + x1 + x1 + x0 + E)**E
*x1**Pi*-1/2i*E*
Pi**x0*(1+-1/2i)*x1*Pi*(1+i)*x0*-1**x1*x0*(x1 + ln 0 + (1/2+-1i) +
(1+-1i) + x1**Pi)*-1i**((1+1/2i)**-1/2i)
+++ OK, passed 10 tests.
```

Definicija *Arbitrary* instance za tip T:

```
instance Arbitrary C where
    arbitrary = oneof [return Pi, return E]

instance Arbitrary T where
    arbitrary = sized genT

chooseQ :: Gen Rational
chooseQ = liftM2 (%) (choose (-1,1))  (choose (1,2))


maxTestCnt = 20000
maxDepth =13
scaleFactor = 2 -- maxTestCnt `div` 2^maxDepth + 1

genT, scaledGenT :: Int -> Gen T
genT n = scaledGenT $ n `div` scaleFactor

scaledGenT 0 = oneof [
     liftM f $ liftM2 (:+) chooseQ chooseQ
   , liftM x $ choose (0,1)
   , liftM TC arbitrary
   ]
scaledGenT n = do
    k <- choose (2,5)
    oneof [
          liftM add $ vectorOf k rec
        , liftM mul $ vectorOf k rec
        , liftM2 pow rec rec
        , liftM ln rec
        ]
   where
   rec =  scaledGenT =<< choose (0, n `div` 2)
```

Bibliografija

[1] http://www.math.wpi.edu/IQP/BVCalcHist/calc5.html

[2] https://people.eecs.berkeley.edu/~fateman/algebra.html

[3] http://stackoverflow.com/questions/7540227/
strategies-for-simplifying-math-expressions

[4] https://docs.haskellstack.org/en/stable/README/

[5] Hackage

Sažetak

Cilj diplomskog rada bio je implementirati sustav za simboličko računanje (CAS) u programskom jeziku Haskell. Iz perspektive CAS-a, ova implementacija ima osnovnu funkcionalnost pojednostavljivanja izraza, te dodatnu funkcionalnost rješavanja sustava linearnih jednadžbi. Iskoristili smo Haskellov načina definiranja tipova (*Algebraic data types*) kako bismo jednostavno implementirali osnovni tip T (term). Zbog visoke razine apstrakcije koju Haskell pruža, omogućeno nam je lako definiranje funkcija poput tmap ili tFoldMap (analogije Haskell funkcija fmap i foldMap, samo nad termima). Preko takvih funkcija prirodno se mogu opisati korisne funkcije i upiti nad termom. Funkcije poput djelovanja nekom funkcijom na svaki čvor terma, ili ispitivanja postojanja čvora tipa Add dvije razine ispod korijena terma. Osim funkcionalnosti vezanih za CAS, kod se sastoji od funkcija koje služe kao definicije za teorijski dio rada. Sva svojstva koja se koriste u definicijama i dokazima teorema, a moguće ih je opisati u Haskell kodu, implementirana su unutar koda. Prednost takvog pristupa je uvid u korektnu definiciju pojedinog svojstva. Za to smo iskoristili Haskellov jezični prevoditelj. Na primjer, svojstvo isNormalFor, koje prima term i vraća Bool vrijednost, se koristi kao dio definicije nekih drugih, složenijih svojstava. Ako u nekom trenutku odlučimo promijeniti tip tog svojstva tako da prima dva terma, jezični prevoditelj nam ne dopušta da napravimo grešku prilikom takve izmjene a da pritom korektno ne izmjenimo primjenu tog svojstva na svakom drugom mjestu gdje se ono pojavljuje. Svojstva koja je moguće definirati u kodu, su dosta ograničena zbog nemogućnosti opisa univerzalnog kvantifikatora nad beskonačnim skupovima, međutim takva mana se ublažava dodavanjem sustava za testiranje (QuickCheck).

Summary

Goal of this graduate thesis was implementation of computer algebra system (CAS) in Haskell programming language. From CAS point of view, this implementation contains basic functionality for term simplification, and extra functionality for solving systems of linear equations. We used Haskell's way of defining data types (*Algebraic data types*) so we could simply implement our main data type T (term). Because of high level of abstraction, which Haskell enables, we can define functions like tmap or tFoldMap (analogies of Haskell's fmap and foldMap, but on data type T). With such functions we can naturally describe useful functions and queries on the T data type. Functions like applying some function on each node of a given term, or querying the existence of a Add-type node two levels bellow the root. Beside the functionality for CAS, code contains functions which serve as definitions for theoretical part of the thesis. All properties which are used in definitions and theorem proofs, and are possible to define in Haskell, are defined in the code. Advantage of such approach is insight in correct property definition. We used Haskell's compiler for such purpose. For example, *isNormal* property of type T $\rightarrow$ Bool, is used as part of some other, more complex properties. If we decide to change the type of this property to T $\rightarrow$ T $\rightarrow$ Bool, compiler won't let us do that unless we accordingly updated all other properties which depend on it. Properties defined in code are quite limited because there is no way to define universal quantification over infinite sets, but we handle this flaw by implementing tests for such properties with QuickCheck.

Životopis

Zovem se Vitomir Čanadi. Rođen sam 19.08.1990. godine u Čakovcu. Pohađao sam Prvu osnovnu školu Čakovec, a nakon toga Graditeljsku školu Čakovec gdje sam završio smjer Medijskog tehničara. Preddiplomski studij Matematike na Prirodoslovno matematičkom fakultetu upisao sam 2008./2009. godinu, a završio 2013./2014. Nakon toga, 2014./2015., upisao sam diplomski studij Računarstva i matematike, koji sam završio 2016./2017.