DEFORESTATSIOON – PUUDE ELIMINEERIMINE FUNKTSIONAALSETES KEELTES

Ain Jahhu
ain244@ut.ee

Programmeerimiskeelte semantika uurimisseminar
Arvutiteatuse instituut, Tartu Ülikool

Detsember 2008

KOKKUVÕTE

Funktsionaalses programmeerimises – või ka üldse programmeerimises – kasutatakse andmete hoidmiseks ja edastamiseks väga palju liste (massiive, jadasid), mille abil seotakse erinevad programmiosad. Need listid võivad aga viia kõrgeks programmi ressursivajadused. Programmi optimeerimiseks piisab juba kasvõi osade listide eemaldamisest.

Käesolevas kirjatükis tutvustatakse ühte üldisemat algoritmi kirjeldust, mis võimaldaks igat tüüpi vahestruktuuride eemaldamist. Samuti kirjeldatakse kahte realisatsiooni Glasgow Haskell kompilaatorile, mis võimaldavad programmi koodi optimiseerida ning eemaldada vaheliste. Esimene lahendus on lihtsam, kuid teine annab rohkem võimalusi.

1. SISSEJUHATUS

Optimiseeriva kompilaatori eesmärgiks on teisendada programmi koodi selliselt, et koodi täitmine oleks väiksema ressursivajadusega. Programmi töö käigus tekkinud andmemassiive – näiteks vaheliste – kasutades on võimalik programmeerijal oma tööd lihtsustada, samas vajavad listid jällegi ressursse. Mõnel juhul, näiteks laisa väärtustamise korral, ei pruugi ruumivajadus ressursina olla väga tähtis, samas vajab ikkagi iga listi elemendi käsitlemine aega ja seega ka ressurssi.

Paljudel juhtudel on võimalik panna koodi kirja nii, et ei kasutata vahepealseid liste – kasutades alternatiivseid funktsioone või programmi teistsugust kirjeldust, näiteks *if* lauseid. Miinuseks on aga, et selle korral võib kaotsi minna funktsioonide kirjelduste selgus ja lühike kirjapilt.

Üheks lahenduseks on *deforestation* [3], algoritm, mis eemaldab suvalised vahepealsed andmestruktuurid (kaasa arvatud listid). Samas on oluline märkida, et mõningatel juhtudel võivad rekursiivsed funktsioonid viia algoritmi täitmise lõpmatusse tsüklisse. Samuti võib rekursiivse funktsiooni korral olla hoopiski vajalik kasutada just optimiseerimata koodi versiooni.

Käesoleva referaadi eesmärgiks on tutvustada puude ja listide elimineerimise algoritmi ning selle realisatsioone Glasgow Haskell kompilaatoris [1, 2, 4], mis kasutavad *deforestation*'i ideed, kuid lahendustes on kasutatud mitmeid lihtsustusi ning rakendatakse funktsioonide *build* ja *foldr* abil tehtavaid teisendusi. Töös kirjeldatakse realisatsioonide põhiomadusi ning tuuakse välja puudused.

2. VAHELISTID

Funktsionaalses programmeerimises on listide kasutamine väga levinud. Seda kasvõi seetõttu, et neis keeltes on olemas palju funktsioone listidega opereerimiseks. Samuti on see väga mugav programmeerijale kuna on võimalik koodi lühidalt ja selgelt kirja panna.

Näiteks kui soovitakse arvutada arvude 1 – n ruutude summat, saab selle kirja panna nii:

sum (*map square* (*upto* 1 n))

Sellise kirjapaneku juures on vahelistid need, mis ühendavad omavahel erinevad funktsioonid. Vahelist [1, 2,…, n] seob omavahel funktsioonid *upto* ja *map* ning list [1, 4, …, n^2] seob funktsioonid *map* ja *sum*. Sellised listid nõuavad aga ressurssi. Laisa väärtustuse puhul ei ole mäluvajadus probleemiks, kuid siiski on vaja iga elemendi puhul aega, et seda luua, uurida ja ära visata.

Sama lõpptulemuse, kuid ilma vahelistideta, annab ka järgmine kood

```
h 0 1 n
where
h a m n = if m > n
              then a
              else h (a + square m) (m + 1) n
```

Kuigi see kood on efektiivsem ei ole see nii selge ja kompaktne.

Teiseks näiteks on funktsioon *all*, mis testib kas kõik listi elemendid rahuldavad etteantud predikaati p. Lühidalt saab selle kirja panna järgnevalt:

all p xs = *and* (*map* p xs)

Funktsioon *map* tekitab vahelisti, mille elemendid (boolean tüüpi) ühendatakse *and* abil kokku. Jällegi saaks funktsiooni kirja panna ilma vaheliste kasutamata:

```
all´ p xs = h xs
      where h []       = True
            h (x : xs) = p x && h xs
```

Ka siin muutub kood segasemaks ja pikemaks.

Siiski annab vahelistide eemaldamine anda märgatava efektiivsuse kasvu. Näiteks arvude 1 – 10 ruutude summat leides tehakse vahelistidega koodi täites 123 tehet aga ilma listideta koodi puhul ainult 73 tehet [2]. See näitab, et juba väikeste arvutuste puhul on olemas üsnagi märgatav vahe funktsiooni efektiivsuses.

Parim lahendus oleks, kui programmeerija saaks kirjutada just sellist koodi nagu talle mugavam on ehk siis kasutades lühikest kirjapilti ning kompilaator teisendab automaatselt koodi efektiivsemale kujule.

Listide eemaldamine tähendab, et ei ole vaja ressursse vahelistide loomiseks ja äraviskamiseks, samuti on listideta koodi puhul võimalik kasutada optimiseerivaid teisendusi, mida muidu ei saanud listide tõttu kasutada.

3. DEFORESTATION ALGORITM

Järgnevas osas kirjeldatud algoritmi juures kasutatakse keelena järgmist grammatika kirjeldust:

t	::=	v	muutuja
		| $c\ t_1 \dots t_k$	konstruktori rakendus
		| $f\ t_1 \dots t_k$	funktsiooni rakendus
		| *case* t_0 *of* $p_1 : t_1 \mid \dots \mid p_n : t_n$	*case* lause
p	::=	$c\ v_1 \dots v_k$	muster

Rakendustes nimetatakse $t_1, \dots, t_k$ argumentideks ning *case* lauses t_0 on selektor ja $p_1 : t_1, \dots, p_n : t_n$ on harud.

Philip Wadler pakkus välja puude eemaldamiseks algoritmi, mille südameks on seitse teisendusreeglit – lisaks on veel neli reeglit märgistatud puude elimineerimiseks, kus puud eelnevalt märgistatakse. Algoritmi tulemusena teisendatakse term – muutuja, konstruktori rakendus, funktsiooni rakendus või *case* lause – t puudeta vormi T[[t]] ning mõlemad kujud peaksid arvutama sama tulemuse [3]. Term on puudeta vormis funktsioonide hulga F suhtes kui term on lineaarne, sisaldab ainult neid funktsioone, mis on hulgas F ning iga funktsiooni konstruktori argument ja iga selektor **case** konstruktsioonis on muutuja. Puudeta vorm tagab selle, et vaheliste ei looda ja ei kasutata korduvaid arvutusi.

(1) $T[[v]] = v$

(2) $T[[c\ t_1 \dots t_k]] = c\ (T[[t_1]]) \dots (T[[t_k]])$

(3) $T[[f\ t_1 \dots t_k]] = T[[t[t_1/v_1, \dots, t_k/v_k]\ ]]$
kus f on defineeritud kui $f\ v_1 \dots v_k = t$

(4) $T[[\text{case}\ v\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n]]$
$= \text{case}\ v\ \text{of}\ p'_1 : T[[t'_1]] \mid \dots \mid p'_n : T[[t'_n]]$

(5) $T[[\text{case}\ c\ t_1 \dots t_k\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n]]$
$= T[[t'_i[t_1/v_1, \dots, t_k/v_k]\ ]]$
kus $p'_i = c\ v_1 \dots v_k$

(6) $T[[\text{case}\ f\ t_1 \dots t_k\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n]]$
$= T[[\text{case}\ t[t_1/v_1, \dots, t_k/v_k]\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n]]$
kus f on defineeritud kui $f\ v_1 \dots v_k = t$

(7) $T[[\text{case}\ (\text{case}\ t_0\ \text{of}\ p_1 : t_1 \mid \dots \mid p_m : t_m)\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n]]$
$= T[[\text{case}\ t_o\ \text{of}$
$p_1 : (\text{case}\ t_1\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n)$
$\dots$
$p_m : (\text{case}\ t_m\ \text{of}\ p'_1 : t'_1 \mid \dots \mid p'_n : t'_n)]]$

Reeglid (1), (2) ja (4) on algne vorm juba puudeta vormis ning komponendid teisendatakse rekursiivselt. Reeglites (3) ja (6) laiendatakse funktsiooni rakendust, tekitades uue termi t, mida

teisendatakse rekursiivselt. Reeglite (5) ja (7) abil lihtsustatakse *case* lauset, mille tulemust teisendatakse jällegi rekursiivselt.

Siiski tuleb meeles pidada, et nende seitsme teisenduse korral on võimalik rekursiivsete funktsioonide teisendamisel minna algoritmi täitmisega lõpmatusse tsüklisse. Näiteks teisendades funktsiooni

$$flip\ (\ flip\ \mathrm{zt}\ )$$

jõutakse algoritmi kasutades seisu

$$\begin{array}{lll} case\ \mathrm{zt}\ of & & \\ \quad Leaf\ \mathrm{z} & : & Leaf\ \mathrm{z} \\ \quad Branch\ \mathrm{xt\ yt} & : & Branch\ (T\ [[\ flip\ (flip\ \mathrm{xt})\ ]]\ )\ (T\ [[\ flip\ (flip\ \mathrm{yt})\ ]]\ ), \end{array} \qquad (*)$$

mis sisaldab algse avaldise kahte ümbernimetatud väljendust. Siin saab uuesti kasutada algoritmi reegleid ning jätkata lõputult. Siinkohal on üheks lahenduseks jälgida algoritmi täitmist ning korduste esinemisel genereerida vastav rekursiivne funktsiooni väljakutse (funktsiooni väljakutse on juba eelnevalt esinenud). Samuti seatakse programmi koodile, mida on võimalik teisendada mõningaid kitsendusi – kõik funktsioonid, mida saab teisendada, peavad olema defineeritud puudeta vormis. Kui kasutada ülaltoodud näite korral kontrolli jaoks funktsiooni h_0:

$$h_0\ \mathrm{zt}\ =\ T\ [[\ flip\ (\ flip\ \mathrm{zt})\ ]]$$

siis jõudes seisu (*) sobivad kaks tähistust *T*[[…]] viimase võrduse parema poolega ning need saab seega vastavalt asendada:

$$\begin{array}{lll} h_0\ \mathrm{zt}\ =\ case\ \mathrm{zt}\ of & & \\ \quad Leaf\ \mathrm{z} & : & Leaf\ \mathrm{z} \\ \quad Branch\ \mathrm{xt\ yt} & : & Branch\ (h_0\ \mathrm{xt})\ (h_0\ \mathrm{yt}) \end{array}$$

See lõpetab teisenduse.

4. LISTIDE EEMALDAMINE

Andrew Gill lahendas probleemi praktilisemal moel, lubades sisendiks suvalist legaalset programmikoodi ning garanteerides kompilaatori töö lõppemise aga samas ei eemaldata kõiki vaheliste. Selleks kasutatakse kahte üldist funktsiooni, millest üks eemaldab ja teine loob liste – *foldr* ja *build*. Funktsiooni *foldr* võiks lühidalt defineerida kui funktsiooni, mis lisab etteantud funktsiooni $\oplus$ iga kahe listi elemendi vahele ning listi lõpus oleva tühja listi märgendi asendab etteantud väärtusega z:

$$foldr\ (\oplus)\ \mathrm{z}\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus \mathrm{z})))$$

Kui defineerida *build* selliselt:

$$\mathrm{build}\ \ g\ =\ g\ (:)\ []$$

siis iga g, k ja z korral,

```
foldr k z (build g) = g k z,
```

kus g tüübiks on

$$g \;:\; \forall\beta.\,(A \to \beta \to \beta) \to \beta \to \beta$$

mingi fikseeritud tüübi A jaoks [1]. Seda võrdust nimetatakse *foldr/build* reegliks.

Funktsioonid, mis lihtsalt eemaldavad liste (kasutades nende elemente) saab kirja panna *foldr* abil ning need, mis ka loovad liste saab kirja panna kasutades lisaks ka funktsiooni *build*. Näiteks:

```
and xs      = foldr (&&) True xs
elem x xs   = foldr (\ a b -> a == x || b)
                    False xs
map f xs    = foldr (\ a b -> f a : b) [] xs

[]          = build (\ c n -> n)
x:xs        = build (\ c n -> c x (foldr c n xs))
concat xs   = build (\ c n -> foldr (\ x y -> foldr c y x) n xs)
filter f xs = build (\ c n -> foldr (\ a b -> if f a then c a b else b) n xs)
```

Esimesed kaks funktsiooni lihtsalt tarbivad listi, kolmas nii tarbib kui ka tekitab listi. Neljas tekitab tühja listi ning kolm viimast on standardfunktsioonide kirjapanek *build* ja *foldr* abil.

Täpsustava näitena *foldr/build* reegli kasutamisest tooks funktsiooni *from* ja *sum* abil. Algselt saab funktsiooni *from* defineerida selliselt

```
from a b = if a>b
           then []
           else a : from (a+1) b
```

aga kasutades abstraktsemat kuju saame

```
from' a b = \ c n -> if a>b
                     then n
                     else c a (from' (a+1) b c n)
```

ning algse funktsiooni saab kirja panna nii:

```
from a b = build (from' a b)
```

Lihtsustades funktsiooni

```
sum (from a b)
    = foldr (+) 0 (build (from' a b))
    = from' a b (+) 0
```

Saadud tulemuses enam vahelisti ei looda.

Kasutades *foldr/build* reeglit saab kasutada selliste funktsioonide kompositsiooni korral, kus üks funktsioon tekitab ja teine kasutab listi.

5. LIST COMPREHENSION – NOTATSIOON

List comprehension notatsioon iseenesest ei ole see midagi muud kui süntaktiline suhkur ehk võimalus jällegi lühidalt kirja panna listidega tehtavaid tehteid. Näiteks paaride listi loomine kahest olemasolevast listist.

$$[(x, z) \mid (x, y_1) \leftarrow r_1,\ (y_2, z) \leftarrow r_2,\ y_1 == y_2]$$

Sellist koodi on võimalik kirja panna ka teisiti nii, et koodi täitmine oleks efektiivsem. Seda eelkõige seetõttu, et üsna tihti kasutatakse *list comprehension* notatsiooni mingi teise funktsiooni, mis kasutab ja eemaldab listi, argumendina – näiteks liidetakse mingile teisele listile juurde.

A. Gill [1] kasutab *list comprehension*'i eemaldamiseks reegleid, mis muudavad tõlkimise lihtsamaks, kuna osa tööst tehakse hiljem *foldr/build* teisendustes. Antudreeglite puhul $\mathcal{TE}$ teisendus tõlgib avaldised rikkamast süntaksist, mis võimaldab ka *list comprehension* kirjeldusi, palju lihtsamasse funktsionaalsesse keelde.

$$\mathcal{TE} :: \text{Expr} \rightarrow \text{CoreExpr}$$

$$\begin{aligned}
\mathcal{TE}\ [[\ [E_1, E_2, \ldots, E_n]\ ]] &= \text{build}\ (\backslash c\ n \rightarrow c\ \mathcal{TE}\ [[\ E_1\ ]]\ (c\ \mathcal{TE}\ [[\ E_2\ ]]\ \ldots\ (c\ \mathcal{TE}\ [[\ E_n\ ]]\ n\ ))) \\
\mathcal{TE}\ [[\ [E_1 .. E_2]\ ]] &= \text{build}\ (\backslash c\ n \rightarrow \text{from'}\ \mathcal{TE}\ [[\ E_1\ ]]\ \mathcal{TE}\ [[\ E_2\ ]]\ c\ n\ ) \\
\mathcal{TE}\ [[\ [E \mid Q]\ ]] &= \text{build}\ (\backslash c\ n \rightarrow (\ \mathcal{TF}\ [[\ [E \mid Q]\ ]]\ c\ n\ ))
\end{aligned}$$

$$\mathcal{TF} :: \text{Expr} \rightarrow \text{CoreExpr} \rightarrow \text{CoreExpr} \rightarrow \text{CoreExpr}$$

$$\begin{aligned}
\mathcal{TF}\ [[\ [E]\ ]]\ c\ n &= c\ \mathcal{TE}\ [[\ E\ ]]\ n \\
\mathcal{TF}\ [[\ [E \mid B, Q]\ ]] &= \text{if}\ \mathcal{TE}\ [[\ B\ ]]\ \text{then}\ \mathcal{TF}\ [[\ [E \mid Q]\ ]]\ c\ n\ \text{else}\ n \\
\mathcal{TF}\ [[\ [E \mid P \leftarrow L, Q]\ ]]\ c\ n &= \text{foldr}\ f\ n\ \mathcal{TE}\ [[\ L\ ]] \\
&\qquad \text{where}\ f\ P\ b = \mathcal{TF}\ [[\ [E \mid Q]\ ]]\ c\ b \\
&\qquad\qquad f\ _\ b = b
\end{aligned}$$

Neid reegleid rakendades on võimalik teisendada näiteks funktsiooni

$$[\ f\ x \mid x \leftarrow \textit{map}\ g\ xs, \textit{odd}\ x\ ]$$

Antud funktsioon tekitab vahelisti *map* funktsiooni tulemuse jaoks. Kasutades ülaltoodud reegleid saame:

$$\begin{aligned}
&\textit{build}\ (\backslash\ c_0\ n_0 \rightarrow \\
&\quad \textit{foldr}\ h\ n_0\ (\textit{map}\ g\ xs) \\
&\qquad \textit{where}\ h\ x\ b = \textit{if}\ \textit{odd}\ x\ \textit{then}\ c_0\ (f\ x)\ b\ \textit{else}\ b)
\end{aligned}$$

Pakkides lahti funktsiooni *map*:

$$\begin{aligned}
&\textit{build}\ (\backslash\ c_0\ n_0 \rightarrow \\
&\quad \underline{\textit{foldr}}\ h\ n_0\ (\ \underline{\textit{build}} \\
&\qquad (\backslash\ c_1\ n_1 \rightarrow \textit{foldr}\ (c_1.g)\ n_1\ xs)) \\
&\quad \textit{where}\ h\ x\ b = \textit{if}\ \textit{odd}\ x\ \textit{then}\ c_0\ (f\ x)\ b\ \textit{else}\ b)
\end{aligned}$$

Rakendades *foldr/build* reeglit on tulemuseks:

```
build (\ c0 n0 ->
        (\ c1 n1 -> foldr (c1.g) n1 xs) h n0
            where h x b = if odd x then c0 (f x) b else b)
```

Kasutades β-lihtsustusi saame:

```
build (\ c0 n0 -> foldr (h.g) n0 xs
        where h x b = if odd x then c0 (f x) b else b)
```

Viimasena saab funktsioonid *foldr* ja *build* lahti pakkida ning peale lihtsustamist on tulemuseks efektiivne avaldis

```
h' xs
   where h' [] = []
         h' (x:xs)      = if odd x'
                          then f x' : h' xs
                          else h' xs
              where x' = g x
```

6. KOMPILAATORI REALISATSIOON

Üks lihtsa lähenemisena realiseeriti *foldr/build* abil puude elimineerimise algoritm [1], [2] Glasgow Haskell'i kompilaatorisse [5]. Katsetused näitasid, et algoritmi rakendamisega saavutati ka märgatav efektiivsuse kasv. Samas on antud realisatsioonil ka mõningad piirangud ja võimalikud edasiarenduskohad. Nimelt funktsioonid – näiteks *tail* ja *foldr1* – millel ei ole harilikku rekursiivset mustrit nõuavad täieliku puude elimineerimise algoritmi rakendamist. Lisaks ei saa *foldr/build* abil lihtsustada funktsiooni *zip.*

Kompilaatori rakendamisel läbitakse järgmised sammud:

1. Peale tüübikontrolli eemaldatakse programmist süntaktiline suhkur.
2. Läbitakse lihtsustuse teisendused – mõned muutujad asendatakse väärtustega, teostatakse β-lihtsustused. Selles sammus veel foldr/build teisendusi ei rakendata.
3. Rakendatakse foldr/build teisendused ning võimalusel lihtsustatakse programmi – näiteks *foldr (:) [] xs* teiseneb kujule *xs*.

Testimiste tulemusena paranesid programmide efektiivsused – siiski mitte väga olulisel määral – kuni 25%. Testitud programmidest suuremal osal ei olnud kas üldse mingit paranemist või siis oli see väga väike (alla 1%).

Samas näiteks kasutades optimiseerivat kompilaatorit 10 lipu paigutamiseks malelauale olid testitulemused sellised: esialgne programm kasutab 179MB ja töötab 24,4 sekundit, kuid teisendatud programm kasutab 36MB ja töötab 8,8 sekundit [1].

7. KOMPILAATORI LAIENDUSED

Lisaks eelnevalt juba realiseeritud lahendusele listide eemaldamiseks programmist [2] on Simon Peyton Jones koos Andrew Tolmach'i ja Tony Hoare'iga realiseerinud lahenduse, mille abil on programmeerijal võimalik täpsustada programmi omadusi, mida kompilaator saab kasutada

jõudluse tõstmiseks [4]. Igat kirja pandud omadust kasutab kompilaator kui ümberkirjutusreeglit. Programmeerijale on antud võimalus suhelda kompilaatoriga ning seeläbi selgitada võimalikke lihtsaid optimeerimisvõimalusi. Sellised kirjeldused pannakse kirja reeglitena nagu näiteks:

```
{-# RULES
        "map/map" forall f g xs.
                map f (map g xs) = map (f . g) xs
#-}
```

Sellise reegli abil oskab kompilaator viia funktsiooni

```
map f (map g xs)
```

kujule

```
map (f . g) xs
```

Mitteoptimiseeriv kompilaator lihtsalt ignoreerib kirjeid "{-# ... #-}". Võtmesõna *RULES* identifitseerib, et tegu on ümberkirjutusreegli definitsiooniga, ""map/map" annab reeglile nime – vajalik diagnostika jaoks – ning *forall* osa kirjeldab reegli kehas olevaid muutujaid – ülejäänud on konstandid nagu selles näites *map*.

Lisaks *RULES* notatsioonile on programmeerija käsutuses *SPECIALISE* kirjeldus, mis julgustab kompilaatorit looma ja kasutama täpsemaid spetsiifilisi funktsioone. Näiteks võimaldavad Haskell'i tüübiklassid selliseid funktsioone nagu

```
invert : : Num elt => Matrix elt -> Matrix elt
```

Selliste funktsioonide korral tekib lihtsama kasutuse korral ebaefektiivsus. Selle vältimiseks on programmeerijal võimalik kirjutada järgmine kirjeldus

```
{-# SPECIALISE
        invert : : Matrix Int -> Matrix Int
#-}
```

Kui kompilaator on selle kirjelduse jaoks koostanud funktsiooni ja nimetanud selle nimega *invert_Int*, siis järgmiseks kindlustab kompilaator, et kõik sobivad funktsioonid *invert* asendataks uue funktsiooniga *invert_Int*. Selleks koostatakse dünaamiliselt reegel, mis võiks välja näha selline:

```
{-# SPECIALISE
        "invert/Int" forall d : : Num Int .
                invert @ Int d = invert_Int
#-}
```

See reegel kirjutatakse GHC vahekeeles "Core". Ka programmeerija poolt kirja pandud reeglid tõlgitakse Core-keelde – tüübikontrolli poolt ning seejärel teisendatakse Haskelli süntaks palju piiratuma Core süntaksi vormi.

Antud realisatsiooni korral on lihtsamate programmide efektiivsuse kasv 5-25%. Siiski ei ole testitud suvalise programmi koodi koos programmeerija poolt lisatud ümberkirjutusreeglitega, kuid arvatavasti võib sellise koodi korral olla efekt isegi suurem.

Näiteks:

```
three_partitions : : Int -> [(Int, Int, Int)]
three_partitions m
        = [ (i, j, k) |  i <- [ 0 . . (m ´div´ 3) ] ,
                         j <- [ i . . (m – i ´div´ 2) ] ,
                         let k = m – ( i + j )
        ]
main = print (length (three_partitions 4000) )
```

See programm kasutab algselt 188 MB mälu, kuid teisendatud variant ainult 16 MB.

8. KOKKUVÕTE

Kui Philip Wadleri kirjeldatud algoritm tutvustab võimalikku lahendust puude eemaldamiseks programmist, siis Andrew Gill ning Simon Peyton Jones on loonud lahendused eelkõige listide eemaldamiseks. Sealjuures on A. Gill'i pakutud variand lihtsam, kuigi pakub vähem võimalusi. Samuti pakub see konkreetset realisatsiooni, mitte ainult teoreetilist lahendust. Simon P. Jones'i pakutav kasutab eelnevat lahendust ning lisab sellele laiendusi ja parandusi dünaamilisuse ja võimalike sisendite suhtes. Üldiselt võib arvata et Jones'i realisatsioon on efektiivsem just seetõttu, et on kasutatud eelmisi lahendusi ning neid on parendatud ja edasi arendatud.

Algoritmi lahenduste täienduseks oleks võimalik lisada realisatsioonile veelgi dünaamilisust, kus programmeerija ei pea maksimaalset efektiivsust silmas pidades kasutama kõrgtaseme operaatoreid nagu *foldr*, vaid kasutades kontekstist sõltuvat otsingut tuvastada tsükleid ja rekursioone. Samuti oleks programmeerijale kasulik saada tagasisidet selle kohta milliseid tema poolt kirja pandud ümberkirjutusreegleid kasutati ja mida mitte.

VIITED

[1] Andrew Gill, Simon Peyton Jones, John Launchbury, *A short cut to deforestation.* FPCA 1993
http://research.microsoft.com/~simonpj/papers/deforestation-short-cut.ps.Z

[2] Andrew John Gill, *Cheap Deforestation for Non-strict Functional Languages.* Glasgow 1996
http://www.dcs.gla.ac.uk/fp/authors/Andy_Gill/thesis.ps.gz

[3] Phil Wadler, *Deforestation: transforming programs to eliminate trees.* TCS 1990
http://homepages.inf.ed.ac.uk/wadler/topics/deforestation.html

[4] Simon Peyton Jones, Andrew Tolmach, Tony Hoare, *Playing by the Rules: Rewriting as a practical optimisation technique in GHC*. 2001
https://eprints.kfupm.edu.sa/58172/1/58172.pdf

[5] The Glasgow Haskell Compiler http://www.haskell.org/ghc