Funktsionaalsed sõltuvused ja tüübipered

Jaak Randmets
jaak.ra@gmail.com

Programmeerimiskeelte semantika uurimisseminar
Arvutiteaduse instituut, Tartu Ülikool

15. detsember 2008. a.

Kokkuvõte

Tüübiklassid programmeerimiskeeles Haskell lubavad programmeerijal defineerida funktsioone, mida on võimalik kasutada hulgal erinevatel tüüpidel, kusjuures funktsiooni definitsioon võib iga tüübi korral erinev olla. Haskell võimaldab deklareerida ainult ühe parameetriga tüübiklasse, kuid loomulikuks laienduseks on mitmeparameetrilisus. Vaatamata sellele, et mitmeparameetrilistele tüübiklassidele on teada hulgaliselt rakendusi ei tööta nendest enamus ebatäpse ning mitmese tüübituletuse tõttu. Üheks mainitud probleemide lahenduseks on laiendada tüübiklasse relatsiooniliste andmebaaside teooriast tuntud funktsionaalsete sõltuvustega. Uuemaks lahenduseks samale probleemile on assotsieeritud tüübid ning, selle mõiste üldistusena, tüübipered.

Käesolevas seminaritöös illustreerime probleeme, mis esinevad mitmeparameetriliste tüübiklassidega ning näidete varal demonstreerime, kuidas nii funktsionaalsed sõltuvused, kui tüübipered need probleemid lahendavad. Lisaks anname võrdleva ülevaate mainitud kahest konkureerivast keele laiendusest ning arutame, kas funktsionaalsete sõltuvuste või tüübiperede standardiseerimine omab tulevikus mõtet.

1 Sissejuhatus

Funktsionaalse programmeerimiskeele Haskell tüübiklassid [14] võimaldavad programmeerijal defineerida funktsioone, mida saab kasutada paljudel erinevatel tüüpidel, kusjuures funktsiooni definitsioon võib olla erinev iga tüübi korral. Näiteks tüübiklass *Eq* esitab võrreldavaid tüüpe, see on tüüpe, mille väärtuste võrdsust on võimalik kontrollida. Sarnaselt esitab tüübiklass *Num* arvulisi tüüpe, mille väärtusi saab muuhulgas liita, lahutada ja korrutada. Mõningateks näideteks *Num* tüübiklassi instantsidest on *Int*, *Double*, kompleksarvud ja ratsionaalarvud. Mainitud ja paljud teised tüübiklassid on defineeritud prelüüdis [8] ning programmeerijal on võimalik uusi tüübiklasse deklareerida ning olemasolevaid klasse laiendada uute andmetüüpidega.

Tüübiklassi *Num* üheks võimalikuks deklaratsiooniks on näiteks:

class $Num\ a$ **where**
$(+), (*) :: a \to a \to a$
$negate\ \ :: a \to a$
...

Toodud deklaratsiooni tuleks lugeda kui "tüüp a kuulub tüübiklassi Num, kui sellel on defineeritud operaatorid $(+), (*)$ ning funktsioon $negate$ korrektsete tüüpidega". Deklareeritud operaatoreid saab kasutada sama nimega nii täisarvudel, ujuvkoma arvudel ning ka näiteks kompleksarvudel. Oluline on tähele panna, et kuigi me räägime tavaliselt üle laaditud funktsioonidest ei pruugi tüübiklassi meetodid olla funktsioonid, tegemist võib olla suvaliste väärtustega.

Järgides *Haskell-98* standardit saame tüübiklassides deklareerida ainult ühe parameetri järgi üle laaditud funktsioone. Teiste sõnadega ei võimalda standard deklareerida enam, kui ühe parameetriga tüübiklasse. Tuleb aga välja, et üheks tüübiklasside loomulikuks laienduseks on mitme parameetrilised tüübiklassid, mis võimaldavad defineerida enam, kui ühe tüübiparameetri järgi *ad-hoc* üle laaditud funktsioone. Vaatamata sellele, et mitmeparameetrilised tüübiklassid on hästi tuntud ning neile on hulgaliselt praktilisi rakendusi, ei ole neid standardiseeritud. Nimelt osutub, et enamus mitmeparameetriliste tüübiklasside rakendusi nõuab keele edasist laiendamist. Tuntud laienduseks on funktsionaalsed sõltuvused [7] ning uuemaks ning hetkel vähe levinud laienduseks on tüübipered [2, 1, 12].

Funktsionaalsed sõltuvused ja tüübipered on konkureerivad laiendused, kuna mõlemad lahendavad sama probleemi ning on väljendusvõimsuselt samaväärsed: funktsionaalseid sõltuvusi kasutava koodi saab teisendada tüübiperesid kasutavaks koodiks ning ka vastupidine on võimalik. Käesolevas seminaritöös anname võrdleva ülevaate funktsionaalsetest sõltuvustest ning tüübiperedest ning arutame, kas oleks mõistlik ühte neist standardiseerida. Järgnevas sektsioonis anname lühiülevaate ühe parameetrilistest tüübiklassidest ning ülejärgmises toome sisse mitmeparameetriliste tüübiklasside mõiste ning kirjeldame nendega tekkivaid probleeme. Me eeldame, et lugeja on tutvunud programmeerimiskeelega Haskell.

2 Tüübiklassid

Käesolevas sektsioonis anname ülevaate ühe parameetrilistest tüübiklassidest. Lugeja, kes tunneb ennast keelega Haskell koduselt võib käesoleva sektsiooni vahele jätta aga lugeja, kellele jääb detaile vajaka võib need leida kas standardist [8] või arvukatest veebist leitavatest õpetustest.

Tüübiklassi deklaratsioon määrab klassi nime ning hulga meetodeid, mida sinna klassi kuuluvad tüübid toetama peavad. Näiteks võrreldavate tüüpide klass Eq võib olla deklareeritud järgnevalt:

class $Eq\ a$ **where**
$(\equiv) :: a \to a \to Bool$

Mõlemal real esinev tüübimuutuja a esitab siin suvalist klassi instantsi. Deklaratsiooni võib lugeda, kui "tüüp a kuulub klassi Eq, kui sellel on defineeritud võrdlemise

funktsioon ($\equiv$) korrektse tüübiga". Paneme tähele, et sama deklaratsioon, kus operaatorile on antud tüüp $b \to b \to Bool$, ei ole eelnevaga samaväärne, kuna ($\equiv$) on toodud juhul universaalselt kvantifitseeritud. Niiviisi muudetud klassideklaratsiooni tüübikontrollija ei aktsepteeri, kuna iga võrdluse kasutus viiks mitmese tüübituletuseni.

Tüübiklassi meetodite tüüpides esinevad tüübimuutujad on kitsendatud vastava klassikontekstiga:

$$(\equiv) :: Eq\ a \Rightarrow a \to a \to Bool$$

Tüübimuutuja a esitab siin suvalist tüüpi, aga predikaat $Eq\ a$ piirab a valikut selliselt, et a peab kuuluma klassi Eq. Me võime mainitud predikaati vaadelda ka kui tingimust $a \in Eq$, mis peegeldab intuitsiooni, et Eq on tüüpide hulk. Kitsendatud tüübid määratakse ka funktsioonidele, mis klassimeetodeid otse või kaudselt mingitel määratlemata tüüpidel kasutavad. Näiteks funktsioonidele:

$$\begin{aligned} &member\ x\ xs = any\ (x \equiv)\ xs \\ &subset\ xs\ ys\ \ = all\ (\lambda x \to member\ x\ ys)\ xs \end{aligned}$$

tuletatakse tüübid:

$$\begin{aligned} &member :: Eq\ a \Rightarrow a \to [a] \to Bool \\ &subset\ \ :: Eq\ a \Rightarrow [a] \to [a] \to Bool \end{aligned}$$

Programmeerijal on võimalus nende funktsioonide tüübid ise määrata. Näiteks *member* funktsioonile on lubatud anda tüüp $Int \to [Int] \to Bool$, seda aga siiski ainult juhul, kui Int kuulub tüübiklassi Eq. Tüübiklasside deklaratsioonide, instantside ning kasutamiste korrektsust kontrollib tüübituletaja kompileerimise ajal. Näiteks ei aktsepteerida koodi $show \circ read$, mis on alati mitmene kuna ei ole võimalik tuvastada millist tüübiklassi instantsi tuleb kasutada.

Klasse on lisaks võimalik jagada hierarhiatesse. Üheks Eq alamklassiks on järjestatavate tüüpide klass Ord, mis võib näiteks defineerida operaatorid ($<$) ja ($\leqslant$):

$$\begin{aligned} &\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where} \\ &\quad (<), (\leqslant) :: a \to a \to Bool \end{aligned}$$

Sümbolit $\Rightarrow$ ei tohi siin lugeda, kui implikatsiooni, täpsem oleks isegi vastupidine: iga klassi Ord instants on ka klassi Eq instants. Alamklasside juures me pikemalt ei peatu, kuid siiski on oluline meeles pidada, et alamklassimine on alati võimalik.

Programmeerijal on võimalus deklareerida tüüpe klassi kuuluvaks ning kompilaatori ülesandeks on kontrollida, et toodud deklaratsioonid oleksid korrektsed ja ei kattuks. Näiteks järgnevad isendideklaratsioonid määravad täisarvud ning paarid klassi Eq:

$$\begin{aligned} &\textbf{instance}\ Eq\ Int\ \textbf{where} \\ &\quad x \equiv y = primEqInt\ x\ y \\ &\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)\ \textbf{where} \\ &\quad (x, y) \equiv (x', y') = x \equiv x' \wedge y \equiv y' \end{aligned}$$

Teise isendideklaratsiooni esimene rida ütleb, et tüüpide a ja b võrreldavus on tarvilik paaride (a, b) võrdlemiseks. Eeldades, et klassile *Eq* on antud vaid need kaks instantsi võime teda vaadelda, kui vähimat hulka, mis rahuldab võrdust $\mathrm{Eq} = \{\mathrm{Int}\} \cup \{(a, b) | a \in \mathrm{Eq}, b \in \mathrm{Eq}\}$. Mitte formaalselt on tegemist lõpmatu hulgaga:

$$\mathrm{Eq} = \{\mathrm{Int}, (\mathrm{Int}, \mathrm{Int}), (\mathrm{Int}, (\mathrm{Int}, \mathrm{Int})), ((\mathrm{Int}, \mathrm{Int}), \mathrm{Int}), \cdots\}$$

3 Mitmeparameetrilised tüübiklassid

Predikaate nagu *Eq a* või vaadelda tingimusena, et a kuulub hulka *Eq*, kuid keele süntaks vihjab, et justkui oleks tegemist funktsiooni rakendamisega. See omakorda vihjab võimalusele rakendada predikaati enam kui ühele tüübimuutujale. Kui R on tüübiklass, siis predikaati $R\ a\ b$ võib intuitiivselt vaadelda kahe kohalise relatsioonina tüüpide vahel, ehk $R\ a\ b$ tähendaks, et tüübid a ja b on relatsioonis R. See on loomulik ühe parameetriliste tüübiklasside üldistus, kuna hulgad on unaarsed relatsioonid. Üldisel juhul võime vaadelda n-parameetrilisi tüübiklasse n-kohaliste relatsioonidena tüüpidel.

Vaatame näitena originaalselt Wadleri ja Blotti poolt välja pakutud klassi [14] alamtüüpimise esitamiseks.

class *Coerce a b* **where**
  coerce :: $a \to b$

Ideeks on instantsieerida *Coerce* ainult sellistel tüüpidel a ning b, kus a tüüpi väärtusi saab kadudeta tüübi b väärtusteks teisendada. Sobilikuks instantsiks oleks näiteks:

instance *Coerce Bool Int* **where**
  coerce False = 0
  coerce True = 1

Tegemist on kaunis kunstliku näitega ning seda heal põhjusel.

Mitmeparameetrilistele tüübiklassidele on avastatud mitmeid praktilisi rakendusi [9], tehnilised detailid on välja töötatud ning paljud keele implementatsioonid toetavad neid. Vaatamata sellele ei sisalda *Haskell-98* [8] standard tuge mitmeparameetrilistele tüübiklassidele. Üheks põhjuseks on see, et mitmed välja pakutud rakendused ei tööta praktikas hästi, nimelt seisneb probleem sageli selles, et Haskell võimaldab ainult defineerida liigselt üldiseid relatsioone tüüpidel. Konkreetsemalt viib mitmeparameetriliste tüübiklasside kasutamine tihti mitmese ning ebatäpse tüübituletuseni ning tüübivigade avastamise viivitamiseni. Kirjeldatud nõrkust peegeldab fakt, et kompilaatoriga `GHC` kaasas olevas baas teekide pakis on ainult kaks mitmeparameetrilist tüüpiklassi ning ka need kasutavad peale mitmeparameetriliste tüübiklasside veel ühte keele laiendust [1].

[1] `FlexibleInstances` – vähendab tunduvalt klassi instantsidele standardi poolt peale pandud piiranguid

Klassikaliseks näiteks on tüübiklass pakkumaks ühest liidest hulgale erinevatele kollektsioonidele. Soovime, et liides toetaks näiteks tühja kollektsiooni konstrueerimist, väärtuste sisestamist ning kollektsiooni kuuluvuse kontrollimist. Teegi realiseerimist võiksime alustada tüübiklassiga:

```
class Collects e c where
  empty  :: c
  insert :: e → c → c
  member :: e → c → Bool
  toList :: c → [e]
```

Tüüp e esitab kollektsioonis salvestatavate elementide tüüpi ning c kollektsiooni enda tüüpi. Mõningateks instantsideks sobiks näiteks:

```
instance Eq e ⇒ Collects e [e] where
  empty = []
  insert = (:)
  ...
instance Ord e ⇒ Collects e (Set e) where
```

Kõik tundub paljulubav, kuid paraku on toodud klassideklaratsioonidega tõsiseid probleeme. Esiteks on funktsiooni *empty* tüüp mitmene:

```
empty :: Collects e c ⇒ c
```

Antud juhul on probleem selles, et muutuja e tüüp jääb määratlemata mistõttu ei ole võimalik tuvastada millist kollektsiooni instantsi kasutada tuleb. Üldisemalt on tüüp alati mitmene, kui selle mingi kontekstis esinev tüübimuutuja ei esine tüübi kehas.

Sellest probleemist on võimalik kõrvale hiilida eemaldades *empty* meetodi klassideklaratsioonist. Vaatamata sellele, et *insert* ega *member* ei oma mitmest tüüpi tekivad siiski kiiresti uued probleemid, mis viivad teisele mitmeparameetriliste tüübiklasside probleemini: tüübivigade teatamise viivitamiseni. Vaatame funktsiooni:

```
f x y c = insert x (insert y c)
```

mis saab tüübi:

```
f :: (Collects a c, Collects b c) ⇒
  a → b → c → c
```

Paneme tähele, et x ning y tüübid võivad olla erinevad kuigi need sisestatakse samasse kollektsiooni üksteise järel. Veelgi hullem on aga see, et funktsioonile g = `f True 'a'` tuletatakse tüüp (*Collects Bool c*, *Collects Char c*) ⇒ $c \to c$, mis annab veateate igal rakendusel. Tüübituletaja ei tohiks g tüüpi korrektseks lugeda ning peaks andma veateate juba selle definitsioonil.

Kollektsioonide klassi on võimalik edukalt realiseerida konstruktorklasse [6] kasutades, nimelt võimaldab Haskell anda klassile parameetrina tüübikonstruktoreid. Üheks konstruktorklasside rakenduseks on näiteks monaadide [13] ning noolte [5]

tüübiklassid. Me omistame siin detailid, kuid ka see lahendus on probleemidega, millest enamus langevad kokku järgmises sektsioonis kirjeldatavate funktsionaalsete sõltuvuste nõrkustega. Lugejal on võimalik konstruktorklasse kasutava lahenduse ning selle probleemidega tutvuda artiklis [7].

4 Funktsionaalsed sõltuvused

Funktsionaalsed sõltuvused on *Haskell-98* standardi laiendus, mis muuhulgas lahendab eelmises sektsioonis kirjeldatud mitmeparameetriliste tüübiklasside probleemid. Tüüpiklasside funktsionaalsed sõltuvused kirjeldas M. P. Jones artiklis [7] andes muuhulgas nendest tunduvalt formaalsema ning detailsema ülevaate, kui meie siin teeme.

Funktsionaalsed sõltuvused lubavad programmeerijal deklareerida mitmeparameetrilise tüübiklassi parameetrite vahel konkreetseid funktsionaalseid sõltuvusi. Näiteks eelmise sektsiooni kollektsioonide näidet saame annoteerida sõltuvusega $c \to e$, mida võib lugeda, kui: "tüübimuutuja c määrab üheselt muutuja e". Peale sõltuvuse annotatsiooni ei muutu meie klassideklaratsioonis miski:

$$
\begin{array}{l}
\textbf{class } \mathit{Collects}\ e\ c \mid c \to e \textbf{ where} \\
\quad \mathit{empty} \quad :: c \\
\quad \mathit{insert} \quad :: e \to c \to c \\
\quad \mathit{member} :: e \to c \to \mathit{Bool} \\
\quad \mathit{toList} \quad :: c \to [e]
\end{array}
$$

Antud näite puhul tähendab sõltuv $c \to e$, et kollektsiooni tüüp määrab ära üheselt selle kollektsiooni elementide tüübi, ehk väärtuse tüübi saame teada kollektsiooni enda tüübist.

Üldsemalt on lubatud klassideklaratsioone annoteerida suvalise hulga funktsionaalse sõltuvusega kujul $x_1 \ldots x_n \to y_1 \ldots y_m$, kus $x_1 \ldots x_n$ ja $y_1 \ldots y_m$ on tüübimuutujad ning $n, m > 0$. Üldisel juhul on interpretatsiooniks, et tüübimuutujad x määravad üheselt ära muutujad y.

Funktsionaalsed sõltuvused võimaldavad meil mitmeparameetriliste klassidega täpsemaid relatsioone kirjeldada. Vaatame näitena kolme kahe parameetriga tüübiklassi:

$$
\begin{array}{l}
\textbf{class } C\ a\ b \\
\textbf{class } D\ a\ b \mid a \to b \\
\textbf{class } E\ a\ b \mid a \to b, b \to a
\end{array}
$$

Esimene deklaratsioon ütleb meile ainult seda, et C on mingi kahe kohaline relatsioon tüüpidel. Teisele näitele peale pandud sõltuvus $a \to b$ ütleb juba, et D ei ole mitte suvaline relatsioon, vaid tegemist on funktsiooniga. Viimase näite kaks sõltuvust kitsendavad relatsiooni veelgi, nimelt E on üksühene funktsiooni.

Tüübikontrollija ülesandeks on nüüd lisaks tagada, et klassideklaratsiooni kehas kirjeldatud sõltuvused on iga instantsideklaratsiooni korral rahuldatud. Näiteks järgnevatel instantsidel ei ole lubatud koos ilmuda, kuigi mõlemad neist on üksinda esinedes korrektsed:

instance *D Bool Int*
instance *D Bool Char*

Minnes tagasi kollektsioonide näite juurde võib tunduda, et funktsiooni *empty* tüüp on mitmene, kuna tüübi *Collects e c* ⇒ *c* vasakul pool esineb muutuja *e*, mis tüübi kehas ei esine. Funktsionaalsete sõltuvuste korral antud juhul probleeme ei teki, kuna *e* on üheselt määratud *c* poolt ning *c* esineb ⇒ märgi paremal pool, seega suudab tüübituleja igal *empty* rakendusele *e* tüübi üheselt tuletada. Seega laiendades keelte funktsionaalsete sõltuvustega ei ole enam nii lihtsalt võimalik mitmeseid tüüpe tuvastada.

Probleeme ei teki ka eelmises sektsioonis toodud näitega:

$$f\ x\ y\ c = insert\ x\ (insert\ y\ c)$$

millele tuletatakse tüüp:

$$f :: Collects\ e\ c \Rightarrow e \to e \to c \to c$$

ning seega ei ole avaldis *f True* `'a'` enam tüübikorrektne.

4.1 Näide: tüübitasemel arvutamine

Üheks funktsionaalsete sõltuvuste praktiliseks rakenduseks on osutunud tüübitasemel arvutamine [4]. Käesolevas alamsektsioonis demonstreerime funktsionaalsete sõltuvuste arvutuslikku võimsust realiseerides tüübitasemel liitmise.

Alustame naturaalarvude esitamisega:

data *Z*
data *S a*

Tüübikonstruktoritel *Z* ja *S* puuduvad väärtuse konstruktorid[2] kuna nendeks ei teki meil sisulist vajadust. Toodud kahte tüübikonstruktorit kasutades saame esitada naturaalarve:

type *One* = *S Z*
type *Two* = *S One*
...

Me teame, et tüübiklassid on sisuliselt relatsioonid tüüpidel ning funktsionaalsed sõltuvused võimaldavad neid relatsioone kitsendada. Eelnevalt mainisime, et funktsioonid on kahe kohalised relatsioonid, mille üks komponent määrab üheselt teise. Sarnaselt saame kahe parameetriga funktsioone vaadelda kui kolme kohalisis relatsioone, mille kaks komponendi määravad kolmanda. Seda meeles pidades on loomulik esitada tüübitasemel liitmist kolme parameetriga tüübiklassi abil:

class *Add a b c* | *a b* → *c* **where**
  $add :: a \to b \to c$
  $add = \bot$

[2] *Haskell-98* standard seda ei luba, siin kasutame keele laiendust `EmptyDataDecls`

Tüübimuutujad a ning b esitavad siin funktsiooni argumente ning c resultaati. Tüübiklass sisaldab meetodit *add* mille abil saame kontrollida kas antud kolmik tõepoolest kuulub relatsiooni *Add*.

Jääb järele klaas instantsieerida:

```
instance Add Z b b
instance Add a b c ⇒ Add (S a) b (S c)
```

Esimene deklaratsioon ütleb, et $0+b=b$ ning teine, et kui $a+b=c$ siis $(1+a)+b=(1+c)$.

Testimiseks deklareerime uue muutuja $t = add$ ning anname tüübikontrollijale ülesandeks kontrollida antud tüübi korrektsust, näiteks $t :: \mathit{Three} \to \mathit{Two} \to \mathit{Five}$ on tüübikorrektne, aga $t :: \mathit{Zero} \to \mathit{One} \to \mathit{Zero}$ ei ole.

Loogilise programmeerimiskeeles Prolog võime naturaalarvude liitmise realiseerida järgnevalt:

```
add(z, B, B).
add(s(A), B, s(C)) :- add(A, B, C).
```

Antud lahendus omab selgeid analooge nähtud funktsionaalsete sõltuvuste lahendusega, ning üldiselt ongi funktsionaalsete sõltuvustega tüüpide tasemel programmeerimine relatsiooniline, mitte funktsionaalne.

Rohkem näiteid ning detaile tüübitasemel funktsionaalsete sõltuvustega arvutamisest võib leida artiklist [4] ning *The Monad.Reader* kaheksandast[3] väljaandest.

4.2 Probleemid

Funktsionaalsed sõltuvused lahendavad mitmeparameetriliste tüübiklassidega tekkinud probleemid, kuid funktsionaalsed sõltuvused ise ei ole probleemideta. Käesolevas alamsektsioonis tutvume mõnede funktsionaalsete sõltuvustega probleemiga.

Läbiva näitena kasutame tüübiklassi efektiivsete vektorite realiseerimiseks. Idee on iga elemendi tüübiga assotsieerida spetsialiseeritud vektori esitus. Näiteks täisarve salvestame *unboxed* massiivis ning paare esitame vektorite paari abil. Seega määrab väärtuse tüüp ära üheselt meie massiivi esituse:

```
class ArrayRep e arr | e → arr where
  index :: arr → Int → e
```

Loomulikult praktilises rakenduses oleks klassimeetodeid rohkem. Omistades realisatsioonid vaatame kahte instantsi:

```
instance ArrayRep Int UIntArr
instance (ArrayRep a arr, ArrayRep b brr) ⇒
  ArrayRep (a, b) (arr, brr)
```

Esimene instants määrab täisarvude esituseks efektiivse *unboxed* täisarvude massiivi, teine aga määrab paaride esituseks vektorite paari.

[3] `http://www.haskell.org/sitewiki/images/d/dd/TMR-Issue8.pdf` (15. detsember 2008. a.)

Abstraktsiooni lekkimine. Funktsionaalsete sõltuvuste tüübikontrollija ei luba meil näiteks defineerida kitsendatud tüübiga abifunktsiooni:

$$indexInt :: ArrayRep\ Int\ arr \Rightarrow arr \to Int \to Int$$
$$indexInt = index$$

Nii `GHC`[4] kui ka `Hugs` ei pea antud tüüpi legaalseks ning ainsaks lahenduseks on funktsioonile anda tüüp $UArray \to Int \to Int$, mis paraku on aga selge abstraktsiooni leke. Tegemist on konkreetsete realisatsioonide probleemidega, millele ei ole lihtsat lahendust teada.

Loetavus. Kõik tüübimuutujad, üle mille on funktsioon *ad-hoc* üle laaditud, peavad esinema selle funktsiooni signatuuris. See muudab tüübid raskesti loetavaks, kuna klassi parameetrite arv võib üpris suureks kasvada. Võrdlevas geneerilise programmeerimise uurimuses [3] on seda mainitud ühe tüütusena.

Relatsiooniline programmeerimine. Väärtuste tasemel on programmeerimine keeles Haskell funktsionaalne, kuid nagu eelmises alamsektsioonis nägime on tüüpide tasemel programmeerimine funktsionaalseid sõltuvusi kasutades relatsiooniline. Oleks mõistlik nõuda, et funktsionaalses keeles oleks nii väärtuste, kui tüüpide tasemel programmeerimine funktsionaalne.

Rahuldava realisatsiooni puudumine. Olemas on vähemalt kolm erinevat realisatsiooni, millest ükski ei ole rahuldav. Iga neist realisatsioonidest paneb klassi ja instantsi deklaratsioonidele peale erinevad ning sageli mitteintuitiivsed või liigselt ranged piirangud tagamaks lahenduvat ning termineeruvat tüübituletust. Näiteks ühes realisatsioonis viib näitena toodud vektorite paaride instants mittetermineeruva tüübituletuseni, kui teises toodud instantsideklaratsiooni ei aktsepteerita üldse.

5 Assotsieeritud tüübid ja tüübipered

Käesolevas sektsioonis anname lühiülevaate tüübiperedest [12]. Esimeses ning teises alamsektsioonis tutvustame vastavalt assotsieeritud tüüpe [2] ning assotsieeritud tüübisünonüüme [1], mis osutuvad süntaktiliseks suhkruks tüübiperedele. Tüübiperedest anname ülevaate kolmandas alamsektsioonis, pärast mida arutame mis funktsionaalsete sõltuvuste probleemid tüübipered lahendavad.

5.1 Assotsieeritud tüübid

Eelmises sektsioonis tõime funktsionaalsete sõltuvuste probleeme välja tuues näitena tüübiklassi massiivide esitamise. Idee oli assotsieerida iga tüübiga spetsialiseeritud massiivide efektiivne esitus, see näide on esindajaks tervele klassile tüübiklasside rakendustele – ise optimeerivatele andmestruktuuridele. Laiendades Haskelli süntaksid võiksime sama tulemuse saavutada järgmiselt:

[4] `GHC` küll lubab seda tüüpi kasutades `FlexibleContexts` laiendust, aga tuletab *indexInt* tüübi siiski valesti

```
data Array Int = IntArray UIntArr
data Array (a, b) = PairArray (Array a) (Array b)
```

Paraku aga ei piisa ainult sellest, kuna meil on vaja *index* funktsioon defineerida erinevalt täisarvude ning paaride korral.

Tüübiklassid lubavad deklareerida tüübi-indekseeritud väärtusi, sama mõiste loomulikuks laienduseks on lubada deklareerida klasside abil ka tüübi-indekseeritud tüüpe. Indekseerimise all mõtleme matemaarilises mõttes indekseeritud hulka või peret. Tüübi-indekseeritud väärtus oleks siis tüüpide järgi indekseeritud hulk. Täpselt see lähenemine on võetud artiklis [2] ning on nimetatud assotsieeritud tüüpideks. Me ei lasku siin detailidesse, vaid anname ülevaate assotsieeritud tüüpidest näidete varal. Alustame eelmises sektsioonis kasutatud massiivide tüübiklassi realiseerimisega assotsieeritud tüüpidega:

```
class ArrayElem e where
  data Array e
  index :: Array e → Int → e
```

Tüübiklassi kehas esinev võtmesõna **data** deklareerib klassiga *ArrayElem assotsieerutud tüübi Array*. Analoogselt võime funktsiooni *index* vaadelda, kui klassiga *ArrayElem* assotsieeritud väärtust.

Klassi instantsieerides peame nüüd andma definitsioonid nii väärtusele *index* kui andmetüübile *Array*. Näitena vaatame jälle tuttavat täisarvude ning paaride instantsi:

```
instance ArrayElem Int where
  data Array Int = IntArray UIntArr
  index (IntArray arr) i = indexUIntArr arr i
instance (ArrayElem a, ArrayElem b) ⇒ ArrayElem (a, b) where
  data Array (a, b) = PairArray (Array a) (Array b)
  index (PairArray a b) i = (index a i, index b i)
```

Funktsiooni *index* tüüp saab olema:

```
index :: ArrayElem e ⇒ Array e → Int → e
```

Konstruktor *IntArray* saab tüübi *UIntArr* → *Array Int* ning *PairArray* tüübi *Array a* → *Array b* → *Array* (a, b).

Paneme tähele, et assotsieeritud tüüpide korral on legaalne ka järgneva:

```
indexInt :: Array Int → Int → Int
indexInt = index
```

Seda küll juhul, kui *ArrayRep* on instantsieeritud tüübi *Int* korral, kuid vastasel juhul tahaksime, et kompilaator tüübivea annaks. Näeme, et assotsieeritud tüübid lahendavad funktsionaalsete sõltuvuste abstraktsiooni lekkimise probleemi.

5.2 Assotsieeritud tüübisünonüümid

Eelmises alamsektsioonis nägime, et tüübiklasse on võimalik loomulikult laiendada andmetüüpidega. Täpselt sama lähenemine on võimalik tüübisünonüümidega [1]. Näitena vaatame tuttavat kollektsioonide klassi, aga see kord kasutame liides realiseerimiseks assotsieeritud tüübisünonüüme:

```
class Collects c where
  type Elem c
  empty  :: c
  insert  :: Elem c → c → c
  member :: Elem c → c → Bool
  toList  :: c → [Elem c]
```

Elem on klassiga *Collects* assotsieeritud tüübisünonüüm. Kollektsiooni klassi instantsieerimisel peame defineerima peale kollektsiooni manipuleerivate funktsioonide ka selle kollektsiooni väärtuse tüübi.

Väga lihtne on instantsieerida kollektsioonide klassi näiteks järjendite jaoks:

```
instance Eq e ⇒ Collects [e] where
  type Elem [e] = e
  empty = []
  insert = (:)
  member = elem
  toList = id
```

Mõningateks näideteks kollektsioonidest võiksid veel olla näiteks andmetüüp *Set* ning räsitabelid:

```
instance Ord e ⇒ Collects (Set e) where
  type Elem (Set e) = e
instance (Collects c, Hashable (Elem c))
   ⇒ Collects (Array Int c) where
  type Elem (Array Int c) = Elem c
```

Assotsieeritud tüübisünonüüme kasutades ei muutu mitmeste tüüpide tuvastamist keerukamaks, nagu funktsionaalsete sõltuvuste korral juhtus. Näiteks väärtuse *empty* tüüpi *Collects c* ⇒ *c* vaadates, on selge, et tegemist ei ole mitmese tüübiga kuna kõik tüübi kontekstis esinevad muutujad esinevad ka selle tüübi kehas. Lisaks on lahendatud probleem loetavusega: funktsioonide tüübid ei sisalda enam liigseid tüübimuutujaid.

5.2.1 Võrdsus kitsendused

Oletame, et tahame realiseerida funktsiooni *sumColl*, mis liidab kokku kõik täisarvude kollektsiooni väärtused. Selgelt ei saa seda järgnevalt realiseerida:

```
sumColl :: Collects s ⇒ c → Int
sumColl c = sum (toList c)
```

Probleem esineb tüübis – mitte kõik kollektsioonid ei salvesta täisarve. Vajame moodus piirata funktsiooni täisarvude kollektsioonidele. Teiste sõnadega peame nõudma, et kollektsiooni väärtuse tüüp oleks *Int*. Selle saavutame *võrdsus kitsendusi* kasutades:

$$\begin{aligned}&sumColl :: (Collects\ c, Elem\ c \sim Int) \Rightarrow c \to Int\\&sumColl\ c = sum\ (toList\ c)\end{aligned}$$

sumColl tüüp on piiratud kollektsioonidele, mille elemendid on *Int* tüüpi. Lastes kompilaatoril funktsiooni tüüp tuletada, saame selleks $(Collects\ c, Num\ (Elem\ c)) \Rightarrow c \to Elem\ c$, mis on tegelikult süntaktiline suhkur tüübile $(Collects\ c, Elem\ c \sim a, Num\ a) \Rightarrow c \to a$.

Veel üheks näiteks on sama tüüpi väärtustega kollektsioonide ühendamine. Ka selleks vajame võrdsus kitsendusi, nimelt:

$$\begin{aligned}&merge\ (Collects\ c, Collects\ c', Elem\ c \sim Elem\ c')\\&\quad \Rightarrow c \to c' \to c'\\&merge\ c\ c' = foldr\ insert\ c'\ (toList\ c)\end{aligned}$$

Kitsendus $Elem\ c \sim Elem\ c'$ nõuab, et kollektsioonide c ning c' elementide tüübid oleksid samad ning tüübikontrollija annab veateate, kui üritame kahte erinevate elementide tüüpidega kollektsiooni ühendada.

5.3 Tüübipered

Tüübipered [12], ehk tüübifunktsioonid, on assotsieeritud tüüpide üldistus. Nimelt võimaldavad tüübipered defineerida globaalseid, mitte ainult klassi lokaalseid, indekseeritud tüüpe.

Meenutades eelmise alamsektsiooni kollektsioonide näidet saame selle ümber kirjutada kasutades tüübifunktsioone järgmiselt:

```
type family Elem c
class Collects c where
  empty   :: c
  insert  :: Elem c → c → c
  member  :: Elem c → c → Bool
  toList  :: c → [Elem c]
```

Esimene rida ütleb, et *Elem* on tüübipere, mis on indekseeritud ühe parameetri c järgi. Kollektsioonide klass kasutab mainitud tüübifunktsiooni. Instantsid saame anda sarnaselt assotsieeritud tüübisünonüümidele:

```
type instance Elem [e] = e
instance Eq e ⇒ Collects [e] where
  empty = []
  insert = (:)
  ...
type instance Elem BitSet = Bool
instance Collects BitSet where
```

Kuna programmeerijal on võimalik tüübiperesid uute tüüpidega laiendada, siis ütleme, et tüübipered on avatud.

Võrdsuse kitsendusi saame ka tüübiperedele peale panna, kuna süntaks ning semantika on sama assotsieeritud tüüpide võrdsuse kitsendustele siis siin me pikemalt ei peatu.

5.3.1 Näide: tüübitasemel arvutamine

Tüübipered võimaldavad väga mugavalt tüübitasemel arvutada:

type family $Add\ n\ m$
type instance $Add\ Z\ m = m$
type instance $Add\ (S\ n)\ m = S\ (Add\ n\ m)$

Tüübid Z ning S on samad, mis funktsionaalsete sõltuvuste sektsioonis. Testimiseks realiseerime jällegi funktsiooni:

$add :: a \to b \to Add\ a\ b$
$add = \bot$

Paneme tähele, et erinevalt funktsionaalsetest sõltuvustest on siin tüüpide tasemel arvutamine funktsionaalne, mitte relatsiooniline.

Antud näite korral on tüübiperede avatus halb, kuna kasutaja võib peret *Add* suvaliselt laiendada. Näiteks tüüp *Add Bool Int* ei oma mõistlikku tähendust, kuid ometi on seda võimalik programmeerijal defineerida. Lahenduseks sellele probleemile on suletud (või piiratud) tüübipered, näiteks sooviksime kirjutada:

kind $Nat = Z \mid S\ Nat$
type family $Add :: Nat \to Nat \to Nat$

Mis nõuab, et tüübifunktsiooni *Add* argumendid ning resultaat oleksid tüübitaseme arvud ning annaks tüübivea vastasel juhul. Suletud tüübiperesid[5] veel realiseeritud ei ole.

5.4 Võrdlus funktsionaalsete sõltuvustega

Eelnevates alamsektsioonides nägime kuidas tüübipered lahendavad kõik eelmises sektsioonis toodud funktsionaalsete sõltuvustega esinevad probleemid:

- Abstraktsiooni lekkimise probleem on lahendatud, me võime funktsioonide tüüpe suvaliselt kitsendada, kus assotsieeritud tüüpide korral see iga realisatsiooni puhul võimalik ei ole.
- Loetavus on tunduvalt paranenud, kuna tüübiklasside parameetrite arv on väiksem ning seetõttu on funktsioonide signatuurides vähem müra.
- Erinevalt funktsionaalsetest sõltuvustest on tüüpide tasemel programmeerimine funktsionaalne.

[5] `http://hackage.haskell.org/trac/ghc/wiki/KindSystem` (15. detsember 2008. a.)

- Kompilaator `GHC` seab tüübiperede instantsidele ning võrdsus kitsendustele peale programmeerijale kergelt mõistetavad kitendused tagamaks ühest ning termineeruvat tüübituletust [12, 11].

Tüübipered on väljendusvõimsuselt samaväärsed funktsionaalsete sõltuvustega, see tähendab, et suvalise programmi, mis kasutab funktsionaalseid sõltuvusi, saab teisendada programmiks, mis kasutab tüübiperesid, ning ka vastupidi. Kuigi tüübiperede esitamine funktsionaalsete sõltuvuste abil osutub mittetriviaalseks [12] ei oma tüübipered siin eeliseid.

Paraku ei asenda tüübipered funktsionaalseid sõltuvusi täielikult. Esineb keerukate parameetrite vaheliste sõltuvustega tüübiklasse, mille esitamine tüübiperedega on liigselt keeruline ning raskesti loetav. Samuti võib programmeerijal olla vajadus tüüpide tasemel just relatsiooniliselt programmeerida, ka sellisel juhul on funktsionaalsed sõltuvused eelistatud. Positiivne on see, et tüübipered ja assotsieeritud tüüpe on võimalik edukalt koos kasutada.

6 Kokkuvõte ning järeldused

Käesolevas töös oleme andnud võrdleva ülevaate funktsionaalsetest sõltuvustest ning tüübiperedest. Lisaks tutvusime mitmeparameetriliste tüübiklassidega tekkinud probleemidega mille nii funktsionaalsed sõltuvused, kui ka tüübipered lahendavad. Jääb järele vaid küsida, et kas oleks mõistlik mitmeparameetrilised tüübiklassid standardiseerida koos tüübiperede või funktsionaalsete sõltuvustega.

Peamiselt raske loetavuse ning rahuldava realisatsiooni puudumise tõttu ei ole funktsionaalseid sõltuvusi praeguses seisundis võimalik standardiseerida. Seepärast taandub küsimus sellele, et kas tüübiperesid on mõtet standardiseerida või mitte. Vaatamata sellele, et tüübipered lahendavad paljud funktsionaalsete sõltuvusteste probleemid oleme jõudnud arvamusel, et liiga vara on sellele küsimusele vastust anda.

Peamisteks probleemideks tüübiperedega on:

- Stabiilse realisatsiooni puudumine. Nimelt on tüübipered realiseeritud ainult kompilaatoris `GHC` ning ka sellel realisatsioonil on puudujääke.
- Puudub piisavalt suur valik tüübiperede praktilistest rakendusest. Kuna tüübipered on vähe testitud, võib välja tulla, et need osutuvad praktikas raskesti kasutatavateks või esineb nendega sügavaid probleeme millest veel teada ei olda.

Ainult aeg näitab kas ilmnevad sügavamad probleemid. Rakenduste vähesus tundub, et saab peatselt lahenduse, kuna juba käesoleva töö kirjutamise hetkel on mitmeid teeke, mis nõuavad kompilaatorilt tüübiperede tuge. Mõningateks juba saadaval olevateks praktilisteks rakendusteks on:

- efektiivne funktsionaalne reaktiivse programmeerimise teek *reactive*
- mitmed tüübitasemel arvutamise teegid
- memoiseerimise teek *MemoTrie*

- *Data Parallele Haskell* [10]

Seega on lootust, et mõlemale potentsiaalsele probleemile leiame vastuse lähemate aastate jooksul esiteks stabiilsemate realisatsioonide ja teoreetilise töö ning teiseks rohkete tüübiperede rakenduste näol.

Viited

[1] M.M.T. Chakravarty, G. Keller, and S.P. Jones. Associated type synonyms. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253. ACM New York, NY, USA, 2005.

[2] M.M.T. Chakravarty, G. Keller, S.P. Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 40, pages 1–13. ACM New York, NY, USA, 2005.

[3] R. Garcia, J. Jarvi, A. Lumsdaine, J.G. Siek, and J. Willcock. A comparative study of language support for generic programming. *ACM SIGPLAN Notices*, 38(11):115–134, 2003.

[4] T. Hallgren. Fun with functional dependencies. In *Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Goteborg University, Varberg, Sweden, Jan*, volume 2001, 2001.

[5] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[6] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61. ACM New York, NY, USA, 1993.

[7] M.P. Jones. Type Classes with Functional Dependencies. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 230–244, 2000.

[8] S.L.P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[9] S.P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[10] S.P. Jones, R. Leshchinskiy, G. Keller, and M.M.T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. 2008.

[11] T. Schrijvers, S.P. Jones, M. Chakravarty, and M. Sulzmann. Type Checking with Open Type Functions. 2008.

[12] T. Schrijvers, M. Sulzmann, S.P. Jones, and M. Chakravarty. Towards Open Type Functions for Haskell. In *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*, pages 233–251, 2007.

[13] P. Wadler. Monads for Functional Programming. *Lecture Notes In Computer Science; Vol. 925*, pages 24–52, 1995.

[14] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM New York, NY, USA, 1989.