# Ülevaade tavalisest ja üldisest zipperist

Rein Raudjärv
reinra@gmail.com

Programmeerimiskeelte semantika uurimisseminar
MTAT.03.204.
Arvutiteaduse instituut, Tartu Ülikool
November 2008

**Kokkuvõte**

Zipper on puhas funktsionaalne andmestruktuur, mis võimaldab hõlpsasti töödelda andmeid, mida saab esitada puu kujul.

Idee on lihtne. Mingit asukohta puus märgib vaatluse all olev alampuu ning tee tema ülemtipust kuni juurtipuni. Seega on viidad antud ahelas võrreldes tavapära puu esitusega vastupidises järjekorras. Kõik navigeerimis- ning muutmisoperatsioonid toimivadki antud asukoha struktuuri peal. Nimi „zipper" tuleneb sellest et mööda andmestruktuuri üles-alla liikumine meenutab luku üles-alla tõmbamist („zipper" on ingl. k. tõmblukk).

Käesolev referaat annab ülevaate kahte tüüpi zipperist. Esmalt vaatleme tavalist zipperit, mis sobib ühte tüüpi tippudega puu esitamiseks. Seejärel anname ülevaate üldisest zipperist, mis sobib puu esitamiseks, kus kõik tipud võivad olla ka erinevat tüüpi.

## 1. Sissejuhatus

Programmeerimisel on sageli tarvis vaadelda mingit puu struktuuri koos tema mingi fookuses oleva alampuuga. Seejuures on vaja fookust nihutada puus nii vasakule, paremale, üles kui alla.

Kui antud operatsioonid peavad olema puhtad funktsioonid (argumendiks antavaid andmestruktuure ei tohi muuta), siis on kõige lihtsam juurtipust käesoleva tipuni viivat ahelat iga operatsiooni korral kopeerida. Sel juhul on iga operatsiooni keerukus logaritmiline [Hu97]. Samas kui puhta funktsiooni nõue ära jätta, siis võiks lihtsalt muuta olemasolevat andmestruktuuri ning kõigi operatsioonide keerukus oleks konstantne. Seega on logaritmiline keerukus kehv saavutus. Käesolevas referaadis anname ülevaate puud esitavast andmestruktuurist, mille korral navigeerimisoperatsioonid on puhtad funktsioonid, mis on samas realiseeritavad konstantse keerukusega. Selle asemel et edasi anda juurtipu viidet anname edasi käesoleva tipu viite.

Idee on lihtne. Mingit asukohta puus märgib vaatluse all olev **alampuu** ning **tee** antud tipust kuni juurtipuni. Seega on viidad antud ahelas võrreldes tavapärase puu esitusega vastupidises järjekorras. Kõik navigeerimis- ning muutisoperatsioonid toimivadki antud asukoha struktuuri peal. Kuna antud andmestruktuuris üles-alla liikumine meenutab luku üles-alla tõmbamist, siis nimetas Huet selle **zipperiks** („zipper" on ingl. k. tõmblukk) [Hu97].

Huet kasutas antud andmestruktuuri teoreemide tõestamise abistaja (*proof assistent*) realiseerimisel. Hiljem on zipperit kasutatud ka näiteks failisüsteemi ZipperFS [Ki05] ning aknahalduri Xmonad realiseerimisel [St07].

Tavalise zipperi realisatsioon, mida Huet tutvustas, on küll lihtne, kuid sellel on kaks peamist puudust. Esiteks nõuab realisatsiooni kirjutamine palju korduvat koodi. Teiseks peavad olema vaadeldava puu kõik tipud ühte tüüpi.

10 aastat hiljem (2007) pakkus Michael Adams korraga mõlemale probleemile lahenduse [Ad07]. Ta tutvustas **üldist zipperit** (*general zipper*), mis on tüübikindel, kuid mille realisatsioon ei sõltu vaadeldavast puust. Selle võimaldamiseks peab üldise zipperi tüüp oma parameetritena kaasas kandma ka vaadeldava puu tippude tüüpe. Antud lahendus on oluliselt keerulisem kui n.ö **tavaline zipper**, kuid selle kasutamine on programmeerija jaoks veelgi lihtsam.

Antud referaadi aluseks on kaks artiklit [Hu97] ning [Ad07], mis tutvustavad vastavalt tavalist ning üldist zipperit. Käesolevas referaadis anname neist kummastki lühikese ül**e**vaate. Autori panus on peamiselt eesti keelde tõlkimine, ümbersõnastamine ning koodinäidete ühtlustamine. Kõik koodinäited on esitatud funktsionaalkeeles Haskell.

Eeldame lugejalt vähemalt algteadmisi programmeerimises ning puu andmestruktuuri tundmist. Üldise zipperi vaatlemisel tulevad kasuks ka süvendatud teadmised funktsionaalkeelest Haskell.

Edasised peatükid on üles ehitatud järgmiselt: Teine peatükk annab ülevaate tavalise zipperi andmetüüpidest, kasutamisest ning realisatsioonist. Kolmas peatükk vaatleb üldise zipperi andmetüüpe, kasutamist ning realisatsiooni. Lihtsuse huvides jätame üldise zipperi puhul vaatluse alt välja puus alla liikumise operatsiooni. Viimane peatükk on kokkuvõttev.

Märkus: Mõisteid **tipp** ja **alampuu** käsitletakse tekstis samas tähenduses.

# 2. Tavaline zipper

Käesolevas peatükis vaatleme tavalise zipperi andmestruktuuri tüüpe, kasutamist ning realisatsioone.

## 2.1. Andmetüübid

Vaatleme esmalt tavalise zipperi poolt kasutatava puu enda andmetüüpi.

```
data Tree a = Item a | Section [Tree a]
```

Mingit tüüpi puu (`Tree a`) on kas vastavat tüüpi element (`Item a`) või siis antud tüüpi alampuude list (`Section [Tree a]`). Seega on antud puu kõik tipud ühte tüüpi ning sisuliselt on tegemist hierarhilise listiga.

Defineerime alampuu **konteksti:**

```
data Context a = ContTop | Cont [Tree a] [Tree a] (Context a)
```

Mingit tüüpi alampuu kontekst on kas tühi (`ContTop`) või siis kolmik (`Cont [Tree a] [Tree a] (Context a)`), mis koosneb antud tipu vasakpoolsetest naabertippudest, parempoolsetest naabertippudest ning ülemtipule vastavast

kontekstist (`Context a`). Igal alampuu kontekstil on seega olemas ka vastava alampuu ülemtipu kontekst, sellel omakorda tema ülemtipu kontekst jne kuni juurtipuni, millel on alati tühi kontekst. Märgime veel, et realisatsiooni efektiivsuse huvides hoiame vasakpoolseid naabertippe alati vastupidises järjestuses. Siis on vasakule liikumine realiseeritav konstantse ajaga.

Kuna kontekst sisaldab eraldi vasak- ning parempoolseid naabertippe, siis on seeläbi konteksti kodeeritud ka puuduva tipu ehk **augu** järjekorranumber oma naabertippude seas.

Mingi alampuu kontekst sisaldab seega kogu puud, kust on antud alampuu välja jäetud. Lisame antud alampuu ning saamegi puu esituse, kus on mingi alampuu fookuses:

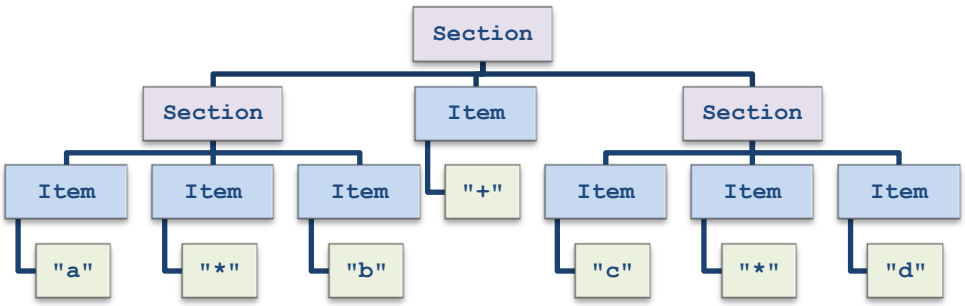
```
data Zipper a = Zipper (Tree a) (Context a)
```

Mingit tüüpi **zipper** on seega paar, mis koosneb antud tüüpi alampuust ning tema kontekstist.

Märgime, et kontekstile vastava alampuu lisamiseks on tõesti vaja eraldi andmetüüpi kuna kontekst ise on hierarhiline.

**Näide:** Vaatleme aritmeetilist avaldist $a \times b + c \times d$, mille võime puu kujul esitada järgmiselt (vt ka joonis 1):

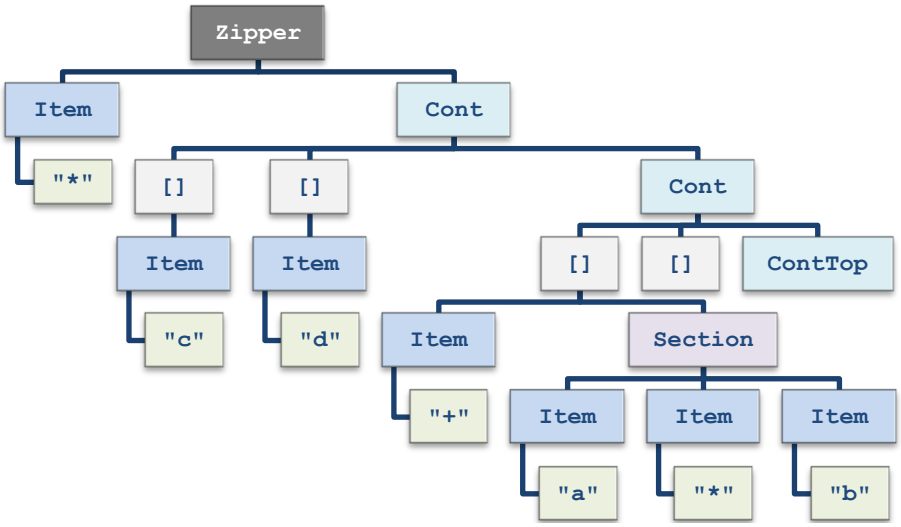
```
myTree = Section [
  Section [Item "a", Item "*", Item "b"],
  Item "+",
  Section [Item "c", Item "*", Item "d"]
]
```




**Joonis 1. myTree struktuur.**

Teisele korrutamismärgile vastava zipperi võime konstrueerida järgmiselt (vt ka joonis 2):

```
myZip = Zipper
  (Item "*")
  (Cont
    [Item "c"]
    [Item "d"]
    (Cont
      [Item "+", Section [Item "a", Item "*", Item "b"]]
      []
      ContTop
    )
  )
```




**Joonis 2. myZip struktuur.**

Zipperi esimene parameeter on fookuses olev alampuu (`Item "*"`). Teise parameetrina anname ette konteksti koos vasakpoolsete (`[Item "c"]`) ning parempoolsete naabertippudega (`[Item "d"]`). Lisaks sellele muidugi ülemkonteksti, kus on kaks vasakpoolset naabertippu (`([Item "+", Section [Item "a", Item "*", Item "b"]]`) ning mitte ühtegi parempoolset naabertippu. Antud konteksti ülemkontekst on aga juba tühi.

Ükski programmeerija ei pea zipperit niimoodi muidugi konstrueerima. Palju lihtsam on luua zipper juurtipu alusel ning seejärel liikuda soovitud tippu. Järgnevalt vaatlemegi vastavaid operatsioone lähemalt.

## 2.2. Navigeerimine

Vaatleme kuidas zipperit kasutades puus vasakule-paremale-üles-alla liikuda. Kõik vastavad funktsioonid on ühte tüüpi:

```
:: Zipper t -> Zipper t
```

Defineerime vasakule liikumise:

```
move_left (Zipper t (Cont (l:left) right up))
        = Zipper l (Cont left (t:right) up)
move_left (Zipper _ (Cont [] _ _))          = error "left of first"
move_left (Zipper _ ContTop)                = error "left of top"
```

Vasakule liikudes eemaldatakse käesolevast kontekstist esimene vasakpoolne naabertipp, mis saab nüüd fookuses olevaks tipuks. Senine käesolev tipp lisatakse aga esimeseks parempoolseks naabertipuks.

Juhul kui vasakpoolseid naabertippe ei ole või fookuses olev tipp on juurtipp, siis antakse viga.

Paremale liikumine on analoogiline:

```
move_right (Zipper t (Cont left (r:right) up))
         = Zipper r (Cont (t:left) right up)
```

```
move_right (Zipper _ (Cont [] _ _)) = error "right of last"
move_right (Zipper _ ContTop)       = error "right of top"
```

Defineerimine nüüd puus ülesse liikumise:

```
move_up (Zipper t (Cont left right up))
       = Zipper (Section ((reverse left) ++ t:right)) up
move_up (Zipper _ ContTop)          = error "up of top"
```

Ülesse liikudes saab fookusesse uus tipp (`Section`), mille alamtipud koostatakse senisest käesoleva tipust ning tema naabertippudest. Kuna vasakpoolseid naabertippe hoitakse kontekstist vastupidises järjekorras, siis tuleb nende järjekord nüüd ümber keerata. Uueks kontekstiks saab vana ülemkontekst.

Juhul kui fookuses oli juurtipp, siis antakse viga.

Alla liikumise defineerime järgmiselt:

```
move_down (Zipper (Section (x:xs)) context)
         = Zipper x (Cont [] xs context)
move_down (Zipper (Section _) _)    = error "down of empty"
move_down (Zipper (Item _) _)       = error "down of item"
```

Alla liikudes saab fookusesse senise tipu (`Section`) esimene alamtipp. Ülejäänud alamtipud saavad uue tipu parempoolseteks naabriteks. Senine kontekst saab uue konteksti ülemkontekstiks.

Juhul kui fookuses oleval tipul ei olnud alamtippe või fookuses oli rippuv tipp (`Item`), siis antakse viga.

Paneme tähele, et alla liikudes saab fookusesse alati esimene alamtipp. Seega kui puus liikuda sammu võrra ülesse ja alla tagasi, siis ei pruugi fookuses olla enam sama tipp vaid esialgse tipu naabertipp.

Funktsioonid puus liikumiseks on kõik konstantse keerukusega välja arvatud üles liikumine, sest siis tuleb fookuses oleva tipu vasakpoolsed naabertipud tõsta vastupidisesse järjekorda.

**Näide:** Kasutades navigeerimise funktsioone võib eelmises näites toodud zipperi konstrueerida järgmiselt:

```
myZip = (move_right.move_down.move_right.move_right.move_down)
        (Zipper myTree ContTop)
```

Siin konstrueerime zipperi kasutades puu juurtippu (`Zipper myTree ContTop`) ning seejärel liigume samm haaval puus alla, paremale, paremale, alla, paremale.

## 2.3. Redigeerimine

Vaatleme esmalt funktsioone, mille abil saab luua uue zipperi, leida fookuses oleva tipu väärtuse ning see uuega asendada. Seejärel uurime funktsioone, mille abil saab puuse lisada uusi tippe ning olemasolevaid eemaldada.

Etteantud puu jaoks zipperi konstrueerimine on triviaalne:

```
begin_zipper :: Tree a -> Zipper a
begin_zipper t = Zipper t ContTop
```

Kuna etteantud puu on juurtipp, siis saab tema kontekstiks tühi kontekst.

Samuti on ilmne, kuidas leida fookuses oleva tipu väärtust ning seda asendada:

```
get_hole :: Zipper a -> Tree a
get_hole (Zipper h _) = h

set_hole :: Tree a -> Zipper a -> Zipper a
set_hole h (Zipper _ context) = Zipper h context
```

Kuna fookuses olev tipp on zipperi esimene parameeter, siis ongi see esimene funktsiooni väärtuseks. Teise funktsiooni puhul konstrueeritakse lihtsalt sama konteksti kuid uue alampuuga zipper.

Vaatleme nüüd puusse uue tipu lisamise funktsioone. Kõik need funktsioonid on ühte tüüpi:

```
:: Tree t -> Zipper t -> Zipper t
```

Defineerime vasakule ning paremale uue tipu lisamise:

```
insert_left l (Zipper t (Cont left right up))
            = Zipper t (Cont (l:left) right up)
insert_left _ (Zipper _ ContTop) = error "insert of top"

insert_right r (Zipper t (Cont left right up))
             = Zipper t (Cont left (r:right) up)
insert_right _ (Zipper _ ContTop) = error "insert of top"
```

Fookuses olev tipp jääb samaks. Uus tipp lisatakse vastavalt vasak- või parempoolseks vahetuks naabriks. Juhul kui fookuses oli juurtipp, siis antakse viga.

Uue alamtipu lisamise defineerimise järgmiselt:

```
insert_down x (Zipper (Section xs) p)
            = Zipper (Section (x:xs)) p
insert_down _ (Zipper (Item _) _) = error "down of item"
```

Fookuses oleval tipule (`Section`) lisatakse vastav uus alamtipp. Kontekst jääb samaks. Juhul kui fookuses oli rippuv tipp (`Item`), siis antakse viga.

Käesoleva tipu puust eemaldamine on aga keerukam:

```
delete :: Zipper t -> Zipper t
delete (Zipper _ (Cont left (r:right) up))
     = Zipper r (Cont left right up)
delete (Zipper _ (Cont (l:left) _ up))
     = Zipper l (Cont left [] up)
delete (Zipper _ (Cont _ _ up)) = Zipper (Section []) up
delete (Zipper _ ContTop)       = error "delete of top"
```

Juhul kui fookuses oleval tipul leidub vahetu vasak- või parempoolne naabertipp, siis eemaldatakse see naabrite hulgast ning temast saab uus käesolev tipp. Kui naabreid enam pole, siis liigutakse samm ülespoole nii, et fookusesse saab tühi tipp ning uueks kontekstiks saab senine ülemkontekst. Juhul kui fookuses oli juurtipp, siis antakse viga.

# 3. Üldine zipper

Tavaline zipperi andmestruktuur on küll lihtne ning efektiivne, kuid tal on kaks suurt puudust:

- kuna kõik tavalist zipperit kasutavad funktsioonid sõltuvad vaadeldava puu tüübist, siis tuleb need vastavalt igale puu tüübile uuesti realiseerida – programmeerija peab kirjutama palju korduvat koodi;

- tavaline zipper ei sobi keerulisemat tüüpi puude esitamiseks kuna kõik puu tipud peavad olema ühte tüüpi.

Michael Adams tutvustas aga 2007. aastal **üldist zipperit** [Ad07], mis lahendab kummagi probleemi jättes samas alles tüübikindluse. Üldine zipper töötab nii puude korral, kus kõik elemendid on sama tüüpi, kui ka keerulisematel juhtudel. Ainsaks tingimuseks on see, et tipud peavad olema `Data` klassi isendid. Kõik tavapärased operatsioonid (`get_hole`, `set_hole`, `move_left`, `move_right`, `move_up`), välja arvatud `move_down`, on täielikud funktsioonid. Nad ei anna kunagi täitmisaegset viga. S.t kui näiteks antud tipul ei ole vasakpoolseid naabreid, siis `move_left` funktsioon ei tüüpu.

Selleks hoiab üldine zipper oma tüübis infot nii fookuses oleva alampuu kui tema konteksti kohta. Seda võimaldavad n.n **üldistatud algebralised andmetüübid** (*Generalized Algebraic Data Types – GADTs* [JWW04]).

## 3.1. Kasutamine

Tavaline zipper ei sobi keerulisemate tüüpide jaoks. Vaatleme näiteks andmetüüpi, mis esitab mingit osakonda:

```
data Dept = D Manager [Employee]
  deriving (Show, Typeable, Data)
data Employee = E Name Salary
  deriving (Show, Typeable, Data)
type Salary = Float
type Manager = Employee
type Name = String
```

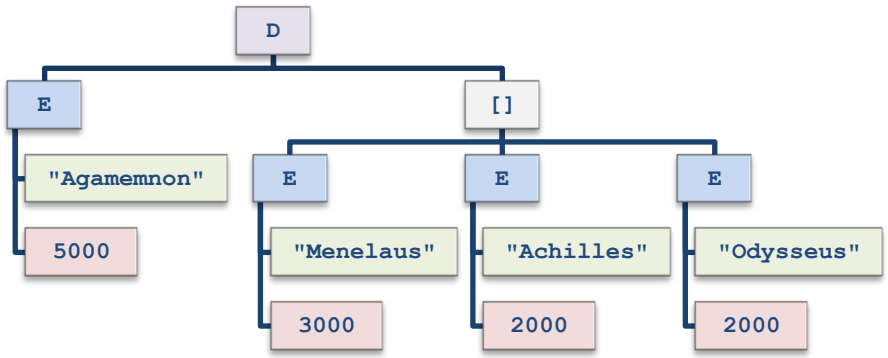
Antud juhul peame eraldi vaatlema `Dept` ning `Employee` tüüpi tippe. Kõik puu tipud ei ole enam ühte tüüpi ning tavalist zipperit ei saa seepärast kasutada. Üldine zipper toetab aga antud puu esitamist. Veelgi enam, programmeerija ei pea ka puus liikumise funktsioone ise valmis kirjutama. Piisab sellest, et kõik puu tippude tüübid pärinevad klassist `Data`, mis kuulub GHC teekide hulka.

Vaatleme näidet osakonda esitavast puust (vt ka joonis 3):

```
company :: Dept
company =
  D agamemnon [menelaus, achilles, odysseus]

agamemnon, menelaus, achilles, odysseus
  :: Employee
```

```
agamemnon = E "Agamemnon" 5000
menelaus = E "Menelaus" 3000
achilles = E "Achilles" 2000
odysseus = E "Odysseus" 2000
```




**Joonis 3. company struktuur.**

Oletame, et meil on antud osakonnas tarvis muuta „Agamemnon-i" nime. Vastava puu muutmiseks peame loome esmalt üldise zipperi:

```
*Main> let g1 = begin_zipper company
*Main> :type g1

g1 :: Zipper (Up (Top, Dept, Top) Top)
```

Zipperi tüüp sisaldab endas fookuses oleva alampuu ning tema konteksti tüüpe. Fookuses oleva alampuu tüübiks on antud juhul `Dept`. Käesolevas kontekstis puuduvad nii vasak- kui parempoolsed naabertipud, seda märgivad vastavalt esimene ja teine `Top`. Seda et fookuses olev tipp on juurtipp märgib aga viimane `Top`.

Zipperi jooksva tipu väärtuse saame `get_hole` funktsiooni abil:

```
*Main> get_hole g1

D (E "Agamemnon" 5000.0)
[E "Menelaus" 3000.0,
E "Achilles" 2000.0,
E "Odysseus" 2000.0]
```

Üldine zipper hoiab enda tüübis fookuses oleva tipu tüüpi, kuid mitte selle alamtippude tüüpe. Viimane sõltub ka sellest, milline konstruktor on fookuses.

Seepärast peame puus alla liikudes ise vastavad alamtippude tüübid täiendavalt ette andma. Üldine zipper kontrollib kas etteantud info oli õige (kasutades üldist tüübikindlast `cast` operaatorit [LJ03]) ja tagastab `Just` või `Nothing` väärtuse vastavalt sellele kas etteantud info oli õige või mitte. Funktsioon `move_down` vajab ilmutatult kogu infot alamtippude tüüpide kohta. See võib aga osutuda sageli väga keeruliseks.

Kui funktsioonile `move_down'` anda ette fookuses oleva tipu konstruktor, tuletab ta vastavad alamtippude tüübid juba ise.

```
*Main> isJust (move_down' g1 D)

True

*Main> let Just g2 = move_down' g1 D
*Main> :type g2

g2 :: Zipper
  (Up (Top -> Employee, [Employee], Dept)
  (Up (Top, Dept, Top)
  Top))

*Main> get_hole g2

[E "Menelaus" 3000.0,
 E "Achilles" 2000.0,
 E "Odysseus" 2000.0]
```

Antud näites võib `g2` tüübist välja lugeda, et fookuses oleva tipu tüüp on `[Employee]`, ainsa vasakpoolse naabri tüüp on `Employee`, parempoolseid naabreid ei ole ning ülemtipu (ülemkonteksti fookuses oleva tipu) tüüp on `Dept`.

Paneme tähele, et üldine zipper on realiseeritud nii, et puus alla liikudes valitakse kõige viimane alamtipp mitte kõige esimene nagu tavalise zipperi puhul. Antud näite puhul peame seega „Agamemnon-ini" jõudmiseks liikuma veel sammu võrra vasakule.

```
*Main> let g3 = move_left g2
*Main> :type g3

g3 :: Zipper
  (Up (Top, Employee, [Employee] -> Dept)
  (Up (Top, Dept, Top)
  Top))

*Main> get_hole g3

E "Agamemnon" 5000.0
```

`g3` tüübist näeme, et fookuses oleva tipu tüüp on `Employee` ning parempoolse naabertipu tüüp `[Employee]`.

Liikudes veel alla ning vasakule jõuame lõpuks „Agamemnon-i" nimeni:

```
*Main> let Just g4 = move_down' g3 E
*Main> :type g4
*Main> let g5 = move_left g4
*Main> :type g5

g5 :: Zipper
  (Up (Top, [Char], Float -> Employee)
  (Up (Top, Employee, [Employee] -> Dept)
  (Up (Top, Dept, Top)
  Top)))

*Main> get_hole g5

"Agamemnon"
```

Fookuses oleva väärtuse muutmiseks kasutame funktsiooni `set_hole`.

```
*Main> let g6 = set_hole "King Agamemnon" g5
```

## 3.2. Andmetüübid

Erinevalt tavalisest zipperist ei sõltu üldise zipperi realisatsioon konkreetsest puu tüübist. Vaadeldava puu tipud ei pea isegi olema sama tüüpi. Ainsaks piiranguks on see, et kõik tippude tüübid oleksid `Data` klassi isendid.

### 3.2.1. Zipper ja kontekst

Üldine zipper koosneb samuti fookuses olevast alampuust ning tema kontekstist.

```
data Zipper path where
  Zipper :: hole
         -> Context (Up (left, hole, right) up)
         -> Zipper (Up (left, hole, right) up)
```

Zipperi tüüp peab garanteerima, et fookuses oleva tipu ning konteksti „puuduva" tipu tüübid oleksid samad.

Konteksti tüüp peab meeles pidama aga naabertippude tüüpe ning ülemtipule vastava konteksti tüüpi.

Kuna konteksti tüüp on üpris keeruline, siis esitame ta algul üldisemal kujul:

```
data Context path where
  ContTop :: Context (Up (Top, a, Top) Top)
  Cont :: Left l (...)
      -> Right r (...)
      -> Context (Up (l_parent, h_parent, r_parent)
                 path)
      -> Context (Up (l, h, r)
                 (Up (l_parent, h_parent, r_parent)
                 path))
```

Kui kõige ülemine kontekst, `ContTop`, välja arvata, siis iga `Context` sisaldab fookuses oleva tipu vasak- ja parempoolseid naabertippe ning vastavat ülemkonteksti.

Need osad, mis on märgitud kolme punktiga, jätsime praegu vaatluse alt välja. Need osad kindlustavad, et ülemtipu tüüp, fookuses oleva tipu tüüp ning tema naabertippude tüübid omavahel sobiksid. Järgnevalt vaatlemegi kuidas naabertippude tüüpe esitada.

### 3.2.2. Vasakpoolsed naabrid

Vaatleme lihtsat andmetüüpi:

```
data Foo = Foo1 Int Char
foo = Foo1 3 'b'
```

Ka selline lihtne avaldis nagu `foo` on üldise zipperi jaoks puu. `3` ja `'b'` on juurtipu `Foo1 3 'b'` alamtipud.

`Foo1` on vaadeldav kui funktsioon, mille tüüp on `Int -> Char -> Foo`.

Mingi tipu naabreid võib seega vaadelda kui antud ülemtipu konstruktori argumente.

Järgnevalt vaatleme tüüpi, mis esitab mingi konstruktori argumente kui tema osalist rakendatust:

```
data Left contains expects where
  LeftUnit :: a -> Left Top a
  LeftCons :: Left c (b -> a)
           -> b
           -> Left (c -> b) a

data Top
  {- no constructors -}
```

Tüübil `Left` on kaks parameetrit `contains` ja `expects`, mis vastavalt esitavad konstruktorile juba rakendatud ning veel rakendamata argumentide tüüpe. Top märgib baasjuhtu, kus konstruktorile pole veel ühtegi argumenti ette antud.

Meie näite korral:

```
*Main> :type LeftUnit Foo1

it :: Left Top (Int -> Char -> Foo)

*Main> :type LeftUnit Foo1 `LeftCons` 1

it :: Left (Top -> Int) (Char -> Foo)

*Main> :type LeftUnit Foo1 `LeftCons` 1 `LeftCons` 'a'

it :: Left ((Top -> Int) -> Char) Foo
```

Näeme, et iga argumendi lisamisel `LectCons` abil liigub vastav tüüp teisest tüübi parameetrist esimesse.

### 3.2.3. Parempoolsed naabrid

Parempoolsete naabrite esitamine on väga sarnane vasakpoolsete naabritega. Erinevuseks on aga see, et selle asemel, et meeles pidada, mis tüüpi alamtippe osaliselt rakendatud konstruktor ootab, tuleb siin meeles pidada, mis tüüpi alamtippe antud tüüp pakub.

```
data Right provides final where
  RightNull :: Right final final
  RightCons :: b
            -> Right a final
            -> Right (b -> a) final
```

(Parameeter `final` ei ole siin praegu oluline. Seda läheb vaja hiljem, kui vaatleme uuesti tüüpi `Context`.)

Kui `Right` pakub täpselt neid tüüpi väärtusi, mida `Left` ootab, siis saabki nende kahe põhjal rakendada vaadeldava konstruktori täielikult. `RightNull`

tähistab seda, et `Right` ei paku ühtegi väärtust. `RightCons` abil saab lisada uue väärtuse, mida `Right` pakub.

Meie näite korral:

```
*Main> :type RightNull

it :: Right final final

*Main> :type 'a' `RightCons` RightNull

it :: Right (Char -> a) a

*Main> :type (1::Int) `RightCons` ('a' `RightCons` RightNull)

it :: Right (Int -> Char -> a) a
```

### 3.2.4. Kontekst

Vaatleme nüüd uuesti tüüpi `Context`. Meil jäi eelnevalt näitamata kuidas on garanteeritud, et ülemkonteksti puuduva tipu tüüp, antud konteksti naabertippude ning fookuses oleva tipu tüübid sobiksid omavahel kokku. Ülemkonteksti puuduva tipu tüübi peab saama konstrueerida kasutades `Left` ja `Right` tüüpe.

```
data Context path where
  ContTop :: Context (Up (Top, a, Top) Top)
  Cont :: Left l (h -> r)
       -> Right r h_parent
       -> Context (Up (l_parent, h_parent, r_parent)
                  path)
       -> Context (Up (l, h, r)
                  (Up (l_parent, h_parent, r_parent)
                  path))
```

Paneme tähele, et `Cont` konstruktori igal argumendil on vähemalt üks ühine parameeter iga teise argumendiga. `Left` ja `Right` puhul on ühiseks parameetriks `r`. See märgib, et `Left` poolt oodatavad tippude tüübid koos fookuses oleva tipu tüübiga peavad olema samad, mis `Right` poolt pakutavad tippude tipud. `Right` ja ülemkontekstil on ühine parmaeeter `h_parent`. Seega peab olema antud konteksti naabertippudele vastava konstruktori tüüp sama, mis ülemkonteksti fookuses oleva tipu tüüp. Seega võime `Left` ja `Right` poolt esitatavate naabertippude ning fookuses oleva tipu põhjal konstrueerida vastava ülemtipu.

## 3.3. Operatsioonid

Vaatleme esmalt puus vasakule-paremale ning ülesse liikumist.

Defineerime vasakule liikumise:

```
move_left :: Zipper (Up (l -> h_new, h_old, r) up)
          -> Zipper (Up (l, h_new, h_old -> r) up)
move_left
  (Zipper h_old (Cont (LeftCons l h_new) r up)) =
  (Zipper h_new (Cont l (RightCons h_old r) up))
```

Vasakule liikumiseks tuleb vasakpoolsetest naabertippudest eemaldada üks `LeftCons` konstruktor ning lisada parempoolsete naabertippude hulka üks `RightCons` konstruktor.

Paremale liikumine on analoogiline:

```
move_right :: Zipper (Up (l, h_old, h_new -> r) up)
           -> Zipper (Up (l -> h_old, h_new, r) up)
move_right
  (Zipper h_old (Cont l (RightCons h_new r) up)) =
  (Zipper h_new (Cont (LeftCons l h_old) r up))
```

Üles liikumise defineerime järgmiselt:

```
move_up :: Zipper (Up child (Up self parent))
        -> Zipper (Up self parent)
move_up
  (Zipper h (Cont l r up)) =
  (Zipper (collapse l h r) up)
```

Siin `collapse` on funktsioon, mis konstrueerib vastavatest tippudest kokku uue ülemtipu.

Alla liikumise funktsioon `move_down` (ning ka `move_down'`) osutub aga äärmiselt keeruliseks ning seda me siinkohal lähemalt ei vaatle. Sellega võib tutvuda [2].

Funktsioonid `begin_zipper`, `get_hole` ning `set_hole` on samad, mis tavalise zipperi puhul:

```
begin_zipper :: h -> Zipper (Up (Top, h, Top) Top)
begin_zipper a = Zipper a ContTop

get_hole :: Zipper (Up (l, h, r) up) -> h
get_hole (Zipper h _) = h

set_hole :: h
         -> Zipper (Up (l, h, r) up)
         -> Zipper (Up (l, h, r) up)
set_hole
  h (Zipper _ context) =
  Zipper h context
```

# 4. Kokkuvõte

Käesolevas referaadis andsime ülevaate tavalisest ja üldisest zipperi nimelisest andmestruktuurist.

Zipper on andmestruktuur, mis võimaldab hõlpsasti töödelda andmeid, mida saab esitada puu kujul. Zipperit on kasutatud nii failisüsteemide (ZipperFS) kui aknahaldurite (Xmonad) realiseerimisel.

Zipperi puhul esitab mingit asukohta puus vastav alampuu ning tema kontekst. Mingile tipule vastav kontekst koosneb tema naabertippudest. Igal kontekstil on viide ülemtipu kontekstile ning nii kuni juurtipuni välja. Kontekstid moodustavad seega ahela, kus võrreldes fookuses oleva alampuuga on puu esitatud „pahupidi”.

Tavaline zipper võimaldab navigeerida mööda puud, mille kõik tipud on ühte tüüpi. Tema realisatsioon on lihtne, kuid see sisaldab palju korduvat koodi. Tavalise zipperi korral võivad kõik navigeerimise funktsioonid anda täitmisaegseid vigu. Kui näiteks liigutakse puus vasakule, kuid vasakpoolseid naabreid ei ole, siis tekib viga.

Üldisel zipperil on tavalise zipperiga võrreldes kaks peamist eelist. Esiteks ei pea kasutaja ise kirjutama oma puu struktuurile vastavat korduvat koodi. Teiseks ei pea olema puu kõik tipud ühte tüüpi. Kasutaja peab ainult tagama, et puu tippude tüübid oleksid Data klassi isendid.

Üldise zipperi korral hoitakse tüübi parameetritena meeles nii fookuses oleva tipu tüüpi, tema naabertippude tüüpe, ülemtipu tüüpi ning tema naabertippude tüüpe jne. Ainsana ei hoita meeles alamtippude tüüpe. Kuna üldine zipper on tüübikindel, siis kui puus alla liikumine välja arvata, ei teki puus navigeerimise käigus programmi täitmisaegseid vigu. Puus alla liikumiseks peab vastavale funktsioonile ette andma lisainfo, mille põhjal tuletatakse käesoleva tipu alamtippude tüübid.

# Viited


[Ad07] Michael Adams. Scrap Your Zippers. http://www.cs.indiana.edu/~adamsmd/papers/scrap_your_zippers/ScrapYourZippers.pdf 2007.

[Hu97] Gerard Huet. The Zipper. Journal of Functional Programming 7 (5): 549-554. 1997.

[JWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004

[Ki05] Oleg Kiselyov. Tool demonstration: A zipper based file/operating system. In Haskell Workshop. ACM Press, September 2005.

[LJ03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices, 38(3):26–37, March 2003.

[St07] Don Stewart. Roll your own window manager: Tracking focus with a zipper. http://cgi.cse.unsw.edu.au/~dons/blog/2007/05/ 17, May 2007.