TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Karl-Martin Uiga

Tüübiohutu FRP teegi uurimine: Grapefruit

Bakalaureusetöö (9 EAP)

Juhendaja(d): Kalmer Apinis

Tartu 2017

Tüübiohutu FRP teegi uurimine: Grapefruit

Lühikokkuvõte:

Käesolev bakalaureusetöö annab lühikese ülevaate funktsionaalsest reaktiivsest programmeerimisest (FRP) ja uurib ühte Haskelli FRP teeki. Selles lõputöös uuritakse programmeerimiskeele Haskell teeki Grapefruit, mille põhilised funktsionaalsused ja võimalused tuuakse välja, tehes seda kolme näidiskoodi selgitamise kaudu. Selle töö lugeja saab hinnata teegi Grapefruit kasulikkust ja kasutajasõbralikkust.

Võtmesõnad:

Grapefruit, FRP, Haskell, programmeerimine

CERCS: P170 – Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Analysing a type-safe FRP library: Grapefruit

Abstract:

The purpose of this Bachelor's thesis is to give a short introduction to functional reactive programming and introduce a Haskell FRP library. This thesis shows different possibilites of a Haskell FRP library called Grapefruit by explaining three sample code's in detail. The reader of this paper can evaluate the efficiency factor and user-friendliness of the Grapefruit library.

Keywords:

Grapefruit, FRP, Haskell, programming

CERCS: P170 – Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus

Blackheath S. ja Jones A. kirjutavad oma raamatus [1], et kuigi maailmas arendatakse pidevalt uut tarkvara ja kõik tundub sujuvat justkui hästi, siis teatud momendil võib arendust tabada seisak. Kogu tarkvara ülesehitus on muutunud liiga keeruliseks ja see omakorda võib mõjutada arendatava tehnoloogia kvaliteeti. Lihtsa lahtri lisamine kasutajaliidesesse võtab ebamääraselt kaua aega, kuigi peaks olema ühe koodirea lisamise vaev. Arendust võiks aidata lihtsustada FRP ehk funktsionaalne reaktiivne programmeerimine. FRP on sõnumitele reageerimisel põhinev süsteem, mis on disainitud puhta funktsionaalprogrammeerimise paradigmale. Hoo sai FRP sisse tänu Elliot C. ja Hudak P. artiklile "Functional Reactive Animation" aastal 1997 [7]. Nüüdseks on sellest möödas 20 aastat, mistõttu võiks FRP olla juba kasutatav ja piisavalt paindlik tänapäeva maailma jaoks. FRP küll ei paranda kõiki programmeerimisest tulenevaid probleeme, kuid aitab ennetada ja parandada liialt keeruliseks muutunud arendusi [1].

Kasutajaliides on interaktiivse tarkvara ülioluline osa, mida on võimalik luua kasutades väga erinevaid programmeerimiskeeli. Nende keelte hulgas on ka programmeerimiskeel Haskell, kus on kasutajaliideste loomiseks mitmeid erinevate omaduste ja võimalustega teeke. Näiteks teek Grapefruit [8], millele antud töö keskendub.

Grapefruit on noole süntaksil baseeruv deklaratiivne teek, mis keskendub kasutajaliidestele [9]. Selles loodavad võimalikud aknad, erinevad vidinad ja neid kontrollivad komponendid suhtlevad signaalide abil [9]. Signaalid on FRP fundamentaalne osa, mis kirjeldavad muutusi üle aja.

Lõputöö eesmärgiks on näidata programmeerimiskeele Haskell teegi Grapefruit erinevaid võimalusi ja eeliseid eesmärgiga luua kasutajaliides. Seda tehakse kolme näidiskoodi varal lahti seletades, millest ühe on kirjutanud teegi autor Wolfgang Jeltsch ning ülejäänud kaks töö autor. Antud näidiskoodid katavad suure osa erinevatest kasutajaliideste elementidest, mis on võimalikud teegiga Grapefruit.

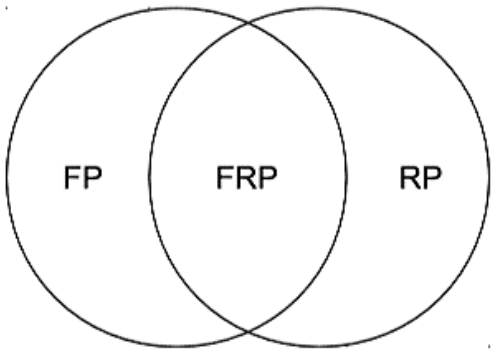
Töö koosneb kolmest peatükist. Esimene peatükk selgitab funktsionaalset reaktiivset programeerimist. Teine peatükk tutvustab programeerimiskeele Haskell teeki Grapefruit, selgitab näidiskoodides kasutatavaid süntakseid ja selgitab ühe teegi autori loodud näidist. Kolmas peatükk selgitab lõputöö autori näidiskoode. Lisadesse on lisatud mõlemad autori poolt loodud näidiskoodid täispikkuses.

2. Funktsionaalne reaktiivne programmeerimine

Alljärgnevas peatükis seletatakse lahti funktsionaalse reaktiivse programmeerimise olemus ja mis on selle seos funktsionaalse- ja reaktiivse programmeerimisega. Lähemalt seletatakse lahti kaks põhilist ajas muutuvat väärtust – sündmus ja käitumine, millele on lisatud ka selgitavad näited koos joonistega.

2.1 Funktsionaalse reaktiivse programmeerimise tuum

Funktsionaalne reaktiivne programmeerimine (edaspidi FRP) on reaktiivse programmeerimise spetsiifiline meetod, mis kasutab funktsionaalse programmeerimise reegleid, täpsemalt struktuuri [1]. FRP ülesanne on muuta kood loetavamaks ja kergemini hallatavaks [1]. FRP koosneb kahest osast – funktsionaalsest ja reaktiivsest programmeerimisest, mida iseloomustab joonis 1:



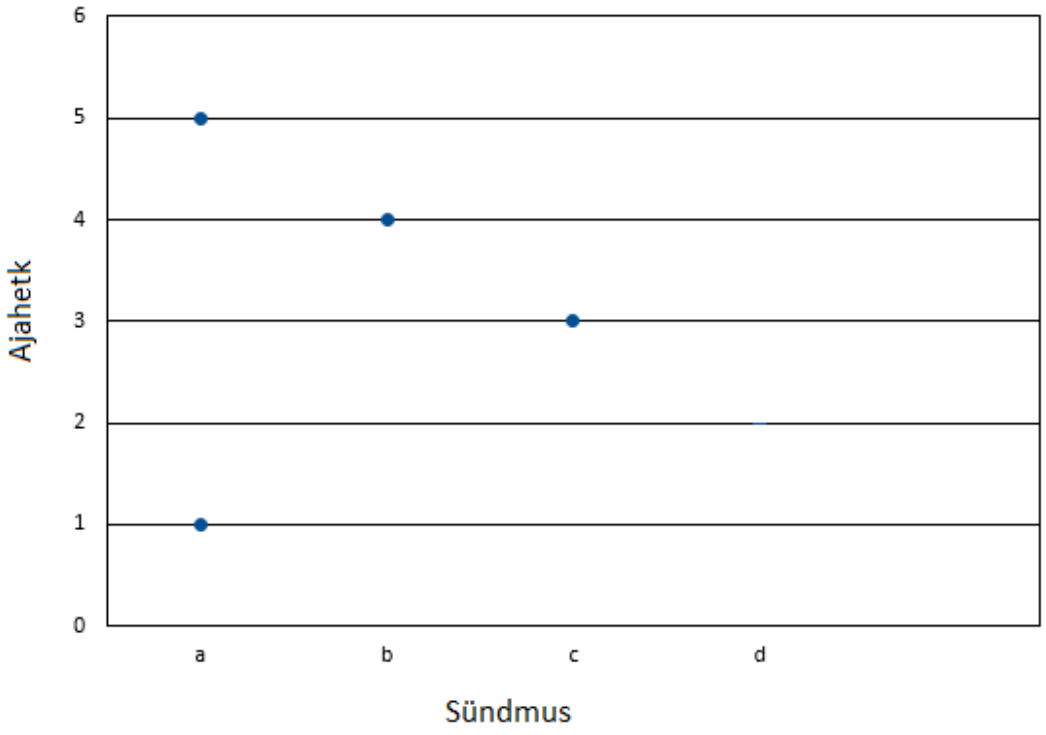
Joonis 1. FP ja RP ühisosa on FRP [1].

- Reaktiivne programmeerimine on lai mõiste, mis tähendab, et programm baseerub sündmustele, käitub vastavalt sisendile ja programmi vaadatakse kui andmevoogu [1].
- Funktsionaalne programmeerimine on deklaratiivne programmeerimise paradigma, kus käskude asemel programmeeritakse deklaratsioonide ja väljendite abil [4]. Funktsiooni väljund sõltub ainult argumentidest mida antud funktsioonile ette antakse (argumentideks võivad olla ka teised funktsioonid) – kutsudes välja funktsiooni sama argumendiga x, mida võib panna kirja mitmel moel (näiteks 2 ja (1 + 1)), annab igal korral sama tulemuse [4].

2.2 Sündmused ja käitumised

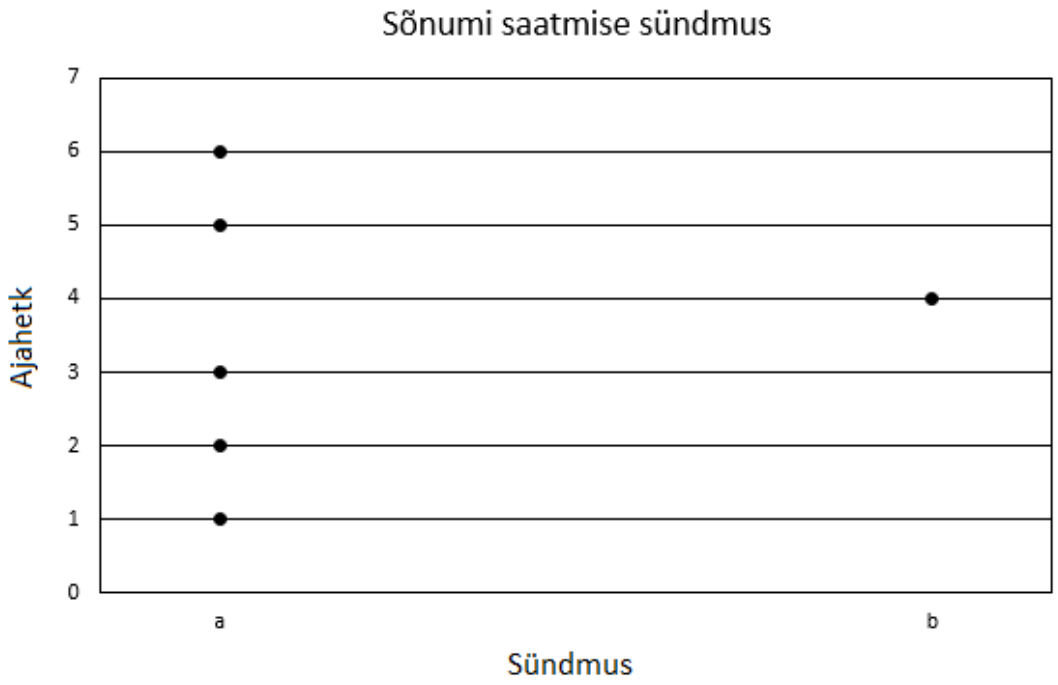
FRP toimub ajas muutuvate väärtustega - sündmused ja käitumised:

- Sündmused on diskreetsed, mingil kindlal ajahetkel toimuvad tegevused, millel on mingi väärtus [1]. Sama liiki sündmuse toimumist mingitel kindlatel ajahetkedel nimetatakse sündmuste vooks [1].



Joonis 2. Sündmuste voogu iseloomustav näide.

Näiteks võime ette kujutada nupusündmuste voogu nii, et ühel ajahetkel on üks sündmus. Võtame seda voogu kui paaride (t, v) järjendit, kus t on sündmuse toimumise ajahetk ja v on sündmuse väärtus ja igal ajahetkel toimub üks kindel sündmus. Näiteks joonisel 2 oleval graafikul on kajastatud sündmuste voog [(1, ‘a’), (3, ‘c’), (4, ’b’), (5, ‘a’)] ehk ajahetkel 1 vajutati klahvi ‘a’, ajahetkel 3 klahvi ‘c’ ja nii edasi.

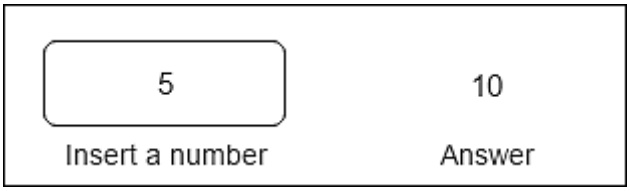


Joonis 3. Sündmust iseloomustav näide sõnumi saatmisest.

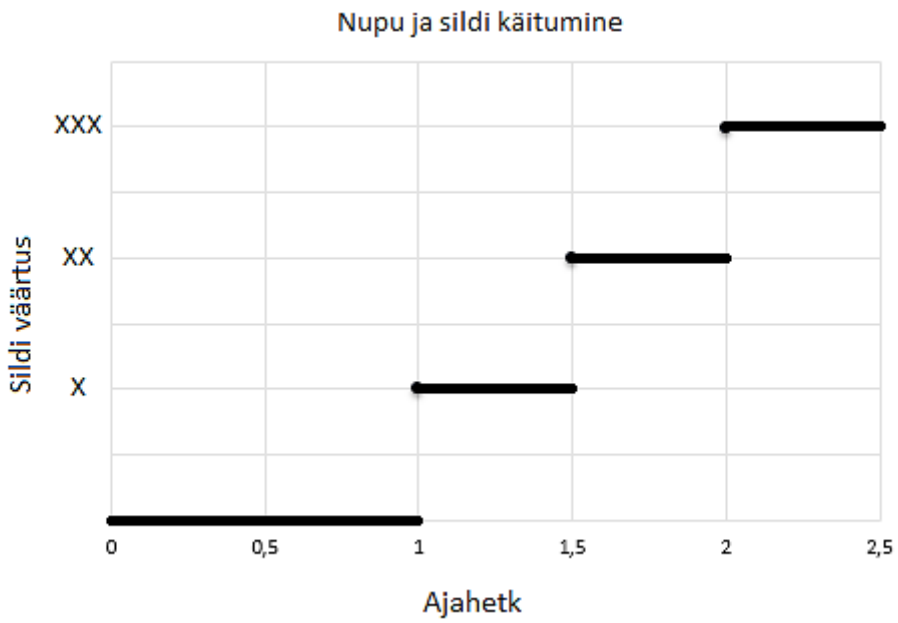
Näiteks võime mõelda sõnumi saatmise sündmust, kus näidatakse erinevate sündmuste toimumist erinevatel ajahetkedel. Seda võib mõelda kui voogu sündmustest (t, v) – *t* on sündmuse ajahetk ja *v* on selle sündmuse väärtus. Joonisel 3 võib sündmust *a* mõelda kui uuendajana, mis toimub ajahetkedel, kus ei toimu sündmust *b*. Juhul kui toimub sündmus *b*, mis on sõnumi saatmist iseloomustav sündmus, siis samal ajahetkel sündmust *a* ei toimu.

- Käitumine – dünaamilised väärtused, mis muutuvad aja jooksul, kuid igal ajahetkel on sellel mingi väärtus [6]. Neid väärtusi saab ette anda funktsioonidele, omavahel kombineerida jne.

Käitumist võib iseloomustada näiteks tekstivälja ja sildi koostöös – tekstivälja kirjutatud arv korrutatakse läbi kahega ning seejärel esitatakse saadud tulemus sildis. Joonisel 4 olev silt on muutuv väärtus, kuid talle on alati midagi omistatud (vaikimisi tühi sõne) - kui tekstivälja kirjutada sõne, jääb silt tühjaks, sest tekstivälja tuleb kirjutada arvuline väärtus.



Joonis 4. Käitumist iseloomustav näide.



Joonis 5. Käitumist iseloomustav näide nupu ja sildiga.

Näiteks iseloomustab käitumist sildi muutumine peale nupuvajutust. Sildile lisatakse peale igat nupuvajutust juurde üks ’X’ ning Joonis 5 kujutab sildi käitumist peale igat nupuvajutust ajahetkel t. Sildil on alati mingi väärtus, olgu see alguses tühi sõne ning kui mingil ajahetkel t vajutatakse nuppu, mistõttu lisandub ka üks ’X’.

3. Grapefruit

Eelnevalt sai väidetud, et Grapefruit on noole süntaksil baseeruv teek, mis keskendub kasutajaliidestele. Sellest tulenevalt seletab antud peatükk lahti kolm põhiliselt kasutusel olevat süntaksit: nooled (*Arrows*), hoidla (*Record*) ja signaalid – neid kasutatakse läbivalt kõigis antud lõputöö näidiskoodides. Antud peatükis on detailselt lahti seletatud ka teegi autori poolt publitseeritud näidis *Converter*, mis on olemasolevatest näidistest lihtsaima ülesehitusega. Näidis iseloomustab sildi ja tekstiredaktori käitumist vastavalt sisendile.

3.1 Näidiskoodides kasutatavad süntaksid

Töös olevates näidetes on läbivalt kasutusel erinevad süntaksid: Funktsiooni konstruktsioon on nool (*Arrow),* andmete hoiustamiseks hoidla (*Record)* ning komponentide vaheliseks suhtluseks kasutatakse signaale.

3.1.1 *Arrow*

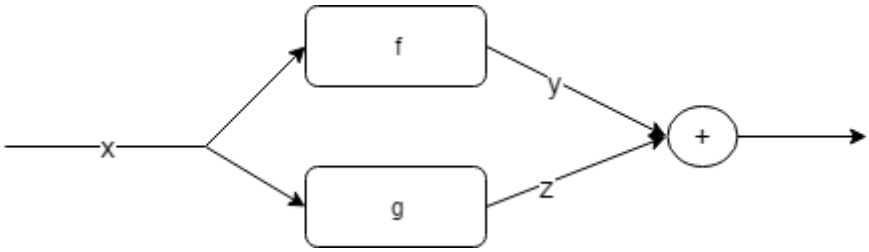
Järgnev alampeatükk tugineb Paterson R. artiklile [5], milles räägitakse, et üldotstarbelised programmeerimiskeeled on enamasti mittepuhtad, mistõttu on võimalik lugeda ja kirjutada andmeid konsooli, kõvakettale või arvutivõrku funktsiooni või meetodi arvutamise käigus. Haskell, seevastu, on puhas funktsionaalne keel, kus tavaline funktsioon (näiteks funktsioon, mis teisendab täisarvu teiseks täisarvuks) ei saa ülejäänud maailmaga suhelda. Välismaailmaga suhtlemist programmeeritakse Haskell'is *Input/Output* monaadi abil.

Seega on funktsiooni tüübi järgi selge, kas see on puhas funktsioon või mitte. Seejuures IO-monaadi arvutusi kutsutakse Haskell's protseduurideks. Näiteks 'Int -> Int' tüüpi funktsioon on puhas ning tema tagastusväärtus sõltub ainult funktsioonile antud argumendist. Tüüp 'Int -> IO Int' on seevastu protseduur, mille tagastusväärtus saab sõltuda nii funktsiooni argumendist kui ka suhtlusest välismaailmaga.

Programmeerimiskeelde Haskell toodi monaad, et arvutada sisendi ja väljundiga (*IO*-monaad), kuid sellele leiti rakendusi ka palju üldisemate tegevuste jaoks. Monaade saab kasutada ka näiteks erinditega programmeerimiseks (*Maybe*-monaad), parsimiseks (*Parser*-monaad) ning lokaalsete muutujatega arvutamiseks (*State*- ja *ST*-monaadid).

John Hughesi poolt loodud nooled (*Arrows)*, mis on üldistus monaadile, võimaldab (potentsiaalselt mittepuhtaid) arvutusi omavahel kombineerida [5]. Noolte intuitiivseks

mõistmiseks on kasulik vaadata noolte graafilist esitust. Joonisel 6 on kujutatud nool, mis saab sisendiks väärtuse x, mis antakse edasi alamarvutustele *f* ja *g* - noole väljundiks on alamarvutuste *f* ja *g* väljundite summa.



Joonis 6. Nool, mis saab sisendiks x ja mille väljundiks on y + z [5].

```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = proc x -> do
                y <- f -< x
                z <- g -< x
                returnA -< y + z
```

Joonis 7. Joonisel 6 kujutatud nool kirjapanduna Haskellis [5].

Järgnev lõik tugineb Haskelli noole kirjeldusele [5], kus joonisel 7 tähistab võtmesõna *pr oc* noole deklareerimise algust, mille järel tuleb anda nimekiri sisenditest (antud koodis *x*) . Noole sisuks on *f*, mis teeb *x*-st *y*-i ja *g*, mis teeb *x*-st *z*-i. Tulemus *y* + *z* tagastatakse peal e konstruktsiooni `*returnA* -<` abil.

Noole süntaksis võib mitme sisendi (või väljundi) kasutamiseks tarvitada `*With*` andmeko nstruktorit. Mitme väärtuse ette andmisel kirjutatakse väärtuste vahele *'With'*.

3.1.2 *Record*

Antud lõputöö näidistes on kasutusel ka *Record* süntaks ning selle seletus tugineb W. Jelt schi artiklil [3]. Seal on kirjeldatud nimetus-väärtus süsteemi – neid paare nimetatakse väljadeks ja neid konstrueeritakse võtmesõnaga *:=*. Saadud väljadest konstrueeritakse andmesturktuur, kus *X* tähistab tühja *Record* tüüpi andmestruktuuri ja selle sees moodusta takse nimekiri elementidest *:&* abil, näiteks õppeaine kohta koostatud *Record*:

```
X     :& Subject    := "Programming"
      :& EAP        :=      6
      :& Lecturer   := "Varmo Vene"
```

Record tüübi kirjeldamisel kasutatakse nimetuse ja tüübi konstrueerimisel *(:::)* tähistust:

```
(X      :& Subject     ::: String
        :& EAP         ::: Integer
        :& Lecturer    ::: String)
```

3.1.3 Signaalid

Alljärgnev materjal on refereeritud W. Jeltschi artiklist [2] kus väidetakse, et signaalid on FRP fundamentaalne osa, mis kirjeldavad käitumisi/muutusi üle aja. Näiteks sildi muutumist nupuvajutuste tagajärel, kus iga nupuvajutusega silti muudetakse. Signaalid jaotuvad alammooduliteks – diskreetsed, segmenteeritud, pidevad ja kasvavad (*incremental*) signaalid.

Diskreetne signaal on järjestus väärtustest, mis implementeerib FRP-s seletatud sündmust. Sellel on väärtus ainult sündmuse toimumise hetkel. Muidu pole väärtus defineeritud. Väärtuse ja ajahetke paari nimetatakse ka sündmuseks (*occurrence*).

Segmenteeritud signaal sarnaneb diskreetsele signaalile, kuid see implementeerib eelnevalt FRP-s käsitletud käitumist. Sellel signaalil on ajas väärtus, kuid seda väärtust saab segmenteeritud signaali puhul muuta vaid siis kui toimub signaali uuendamine (*update points*). Diskreetse signaaliga võrreldes on segmenteeritud signaalil alguspunktis alati mingi väärtus.

3.2 Teegi autori näidis Converter

Tegemist on teegi Grapefruit [8] autori Wolfgang Jeltschi poolt loodud näidiskoodiga, mis demonstreerib erinevate kasutajaliidese elementide (tekstiredaktor ja silt) tööd ja ajas muutuvate muutujate esitamist sildil. Tekstivälja tuleb kirjutada numbriline väärtus, misjärel kuvatakse alumisele tekstiribale selle binaarväärtus. Kui aga kirjutada vale väärtus (nt. „t534“ – ei sobi, kuna „t“ ei ole numbriline väärtus), siis kuvatakse tühi sõne.

Joonis 8. Converter.hs

Näide algab *mainCircuit* funktsiooniga, mille signatuur on esitatud joonisel 9.

```
mainCircuit :: (BasicUIBackend uiBackend) =>
                    UICircuit Window uiBackend era () (DSignal era ())
```

Joonis 9. Funktsiooni mainCircuit signatuur [8].

Joonisel 9 olevas signatuuris on *BasicUIBackend uiBackend* tüübi piiraja, mis võimaldab kasutada erinevaid kasutajaliidese elemente – *lineEditor, label* ja *window*:

- *lineEditor* – kast, kuhu on võimalik teksti sisestada.
- *label* – silt, kus on võimalik teksti esitada.
- *window* – aken, kuhu saab võimalikud kasutajaliidese elemendid lisada.

UICircuit on vooluring, kus ringlevad sõnumitevood, mis koosnevad kasutajaliidese elementidest nagu aken (*Window*) või vidin (*Widgets)*. Vooluringil on võime välise maailmaga suhelda (I/O), seega sealt tulenev sisend ja väljund liidetakse samuti olemasolevasse ringi.

Vooluringi signatuur on *UICircuit item uiBackend era input output*:

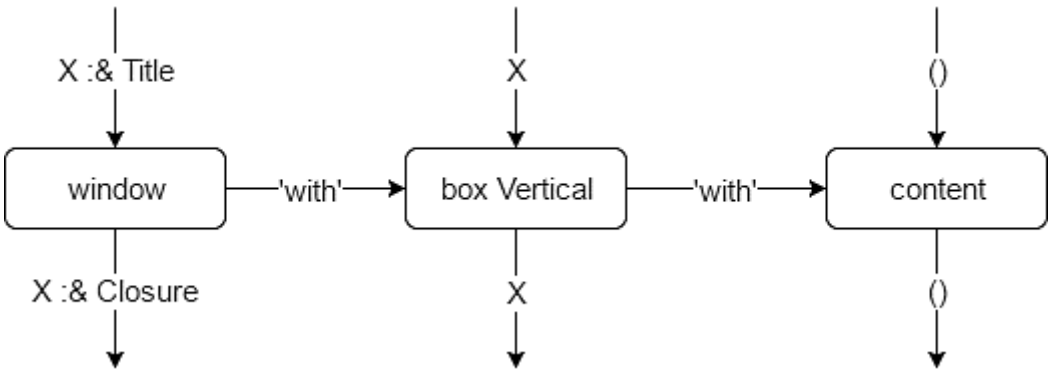
- Parameeter *era* tähistab aja intervalli, mille jooksul loodud *Circuit* eksisteerib.
- Parameeter *item* on kas *Window* või *Widget* tüüpi, kuhu saab edaspidi lisada erinevaid kasutajaliidesega seonduvaid elemente.

Antud näites saab *mainCircuit* sisendiks väärtuse *()* ning väljundiks on *closure*, mis on *DSignal era ()* tüüpi väärtus ja annab teada akna sulgemisest.

Joonisel 10 luuakse nool, kus sisend pole oluline ning väljundiks on *closure ::: DSignal ’Of’ ()* tüüpi väärtus.

```
mainCircuit = proc _ -> do
        X :& Closure := closure `With` X `With` _  <- mainWindow
                                    -< X :& Title := mainWindowTitle
                                                  `With` X `With` ()
        returnA -< closure where
        mainWindow      = window `with` box Vertical `with` content
        mainWindowTitle = pure "Converter"
```

Joonis 10. mainCircuit funktsiooni [8].



Joonis 11. mainCircuit funktsiooni skeem.

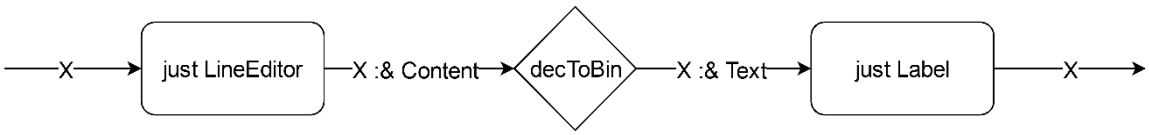
Loodava akna sisu on muutujas *content*, kus on *window*, mille skeemi näeme joonisel 11. Sisu saame anda *with* andmekonstruktoriga, mis muudab tavalise kasti kasutajaliidese komponendiks. Komponendiks anname *UIItem Box*, millele sisestame omakorda sisu *with* andmekonstruktoriga. Saadud *box* annab väljundiks ja võtab sisendiks tühja hoidla *X*.

Muutuja *mainWindow* tahab sisendiks saada *X :& (Req Title ::: (SSignal `Of` String))* tüüpi väärtust, mis tuleb sisse *X: Title := mainWindowTitle* kaudu ning sõne tüüpi signaali saame funktsiooni *pure* abil. Funktsioon *pure* võtab igat tüüpi väärtuse ja paneb selle algvormi, kuid viide algsele väärtusele säilib.

UIItem Box saab oma sisu uuest funktsioonist *content*, mille lähtekood on välja toodud joonisel 12 ning funktsiooni skeem joonisel 13. Sellel on sarnane signatuur *mainCircuit* funktsiooniga, kuid ainuke erinevus on *item* parameetris, kus nüüd on *Window* asemel *Widget*.

```
content :: (BasicUIBackend uiBackend) =>
                                    UICircuit Widget uiBackend era () ()
content = proc _ -> do
        X :& Content := decimal <- just lineEditor -< X
        X                       <- just label      -< X :& Text :=
                                                      decimalToBinary <$> decimal
        returnA -< ()
```

Joonis 12. Funktsiooni content signatuur ja sisu [8].



Joonis 13. Funktsiooni content skeem.

Tahame luua vidinat *Widget*, mis on omakorda eelnevalt *mainCircuit* funktsioonis loodud akna sees (*widget* peab olema *window* sees). Sisendiks ja väljundiks on mõlemal juhul väärtus *()* ning ka selles funktsioonis kasutatakse eelnevalt selgitatud noole konstruktsiooni.

Luues lipikut konstrueeritakse *lineEditor* ning *just* funktsiooniga tehakse *Brick* elemendist *UIItem* element, mis omakorda annab väljundiks *(X :& Content :::: (SSignal 'Of' String))* tüüpi väärtuse ning see sobitatakse mustriga *X :& Content := decimal.* Sisendiks nõuab *lineEditor* aga tühja hoidlat, mis sobitatakse väärtusega *X*.

Konstrueeritakse *label*, mille väljundiks on tühi hoidla ehk *X* ning sisendiks nõuab *X :& (Req Text ::: (SSignal `Of` String))* tüüpi väärtust. See sobitatakse *X :& Text := decimalToBinary <$> decimal* abil, kus muutuja *decimal* tuleb *lineEditor* väljundist.

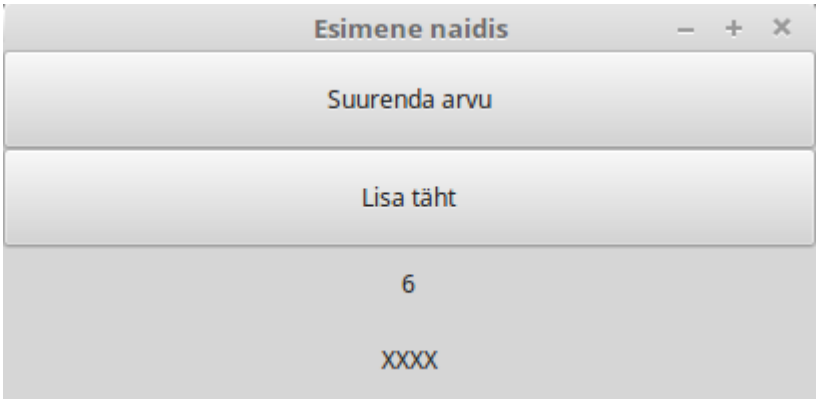
Funktsioon *decimalToBinary <$> decimal* annab tagasi *String* tüüpi väärtuse (*label* nõuab sisendiks tüüpi *String)*. Funktsioonid *decimalToBinary* ning *numberToReverseBinary* muudavad *lineEditor* väärtuse vastavaks binaarnumbriks – kui väärtus on õigel kujul (vastavad kontrollid on tehtud funktsioonide sees), siis tagastab see funktsioon sõne, mis esitatakse sildil.

4. Autori näidiskoodid

Alljärgnevas peatükis on kaks lõputöö autori poolt kirjutatud näidiskoodi, mis on detailideni lahti seletatud. Mõlemas näidises on kasutusel peatükis 3.1 kirjeldatud süntaksid. Esimene näidis on intuitiivsem – kaks nuppu, mida vajutades suureneb kas arv ühe võrra või lisatakse sõnele ‘X’. See iseloomustab eelnevalt peatükis 2 seletatud käitumist. Teine näidis on mõnevõrra pikem, mis näitab hulkade ja signaalide omavahelist tööd.

4.1 Example1

Joonisel 14 on tegemist töö autori loodud näitekoodiga, mis näitab erinevate kasutajaliidese elementide ja segmenteeritud signaali kasutamist. Näites on kaks nuppu ja silti; ülemine nupp suurendab esimese sildi loendurit ühe võrra ning alumine nupp liidab teise sildi sõnele ühe ’X’ juurde, elemendi või arvu lisamiseks saadakse segmenteeritud signaal.



Joonis 14. Example1.hs

```
mainCircuit :: (BasicUIBackend uiBackend) =>
                   UICircuit Window uiBackend era () (DSignal era ())
mainCircuit = proc _ -> do
        X :& Closure := closure `With` X `With` _  <- mainWindow
                            -< X :& Title := pure "Esimene naidis"
                                                  `With` X `With` ()
        returnA -< closure where

        mainWindow  =  window `with` BasicUIBackend.box Vertical
                                                  `with`              boxContent
boxContent :: (BasicUIBackend uiBackend) =>
                    UICircuit Widget uiBackend era () ()
boxContent = proc _ -> do
    rec let
            numberCounter = show <$> SSignal.scan 0
                                     (\xs () -> (1 + xs)) addNumb

            letterCounter = SSignal.scan ""
                                     (\xs () -> ('X' :xs)) addLetter


        X :& Push := addNumb      <- just pushButton
                                  -< X :& Text := pure "Suurenda arvu"
        X :& Push := addLetter    <- just pushButton
                                  -< X :& Text := pure "Lisa täht"
        X <- just label -< X :& Text := numberCounter
        X <- just label -< X :& Text := letterCounter

    returnA -< ()
```

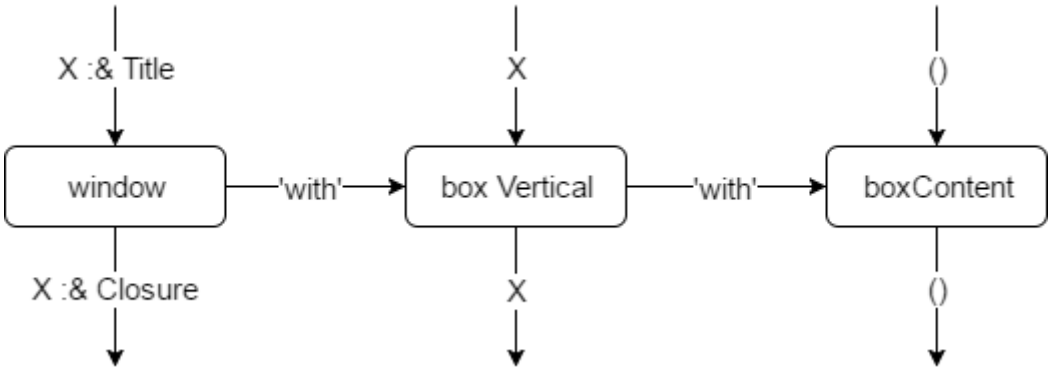
Joonis 15. Esimese näidise lähtekood (täispikkuses on leitav lisas II) .

Näide algab *mainCircuit* funktsiooniga, mille konstruktsioon on toodud välja joonisel 15. Signatuuris olev *BasicUIBackend* võimaldab kasutada vajalikke kasutajaliidese elemente: *Box, Label* ja *pushButton*. Sisendiks nõuab *mainCircuit* funktsioon väärtust *()* ning annab väljundiks *Closure ::: DSignal era ()* tüüpi väärtuse.

Funktsioonis *mainCircuit* luuakse nool, kus sisendi tüüp pole oluline, kuid väljundiks on *Closure ::: DSignal 'Of' ()* vastavalt signatuurile. Muutuja *mainWindow* all olev *window* ehk aken nõuab sisendiks *X :& (Req Title ::: (SSignal `Of` String))* tüüpi väärtust. See sobitatakse *X :& Title := pure "Esimene näidis"* kaudu, kus *pure* funktsiooniga saame vajaliku tüüpi signaali. Väljundiks nõuab *window* tüüpi *X:& (Closure ::: (DSignal 'Of' ()))*, mis sobitatakse *X :& Closure := closure* kaudu.

Aknale (*window)* antakse sisendiks *BasicUIBackend.box*, mis sisestatakse andmekonstruktoriga *'with'*. Sisendi ja väljundi väärtuseks *box* (kast) puhul on hoidla *X*, kuid saamaks vajalikke silte ja nuppe on omakorda kastile antud sisuks *boxContent*, mille sisu on välja toodud joonisel 15, kasutades andmekonstruktorit *'with'*.

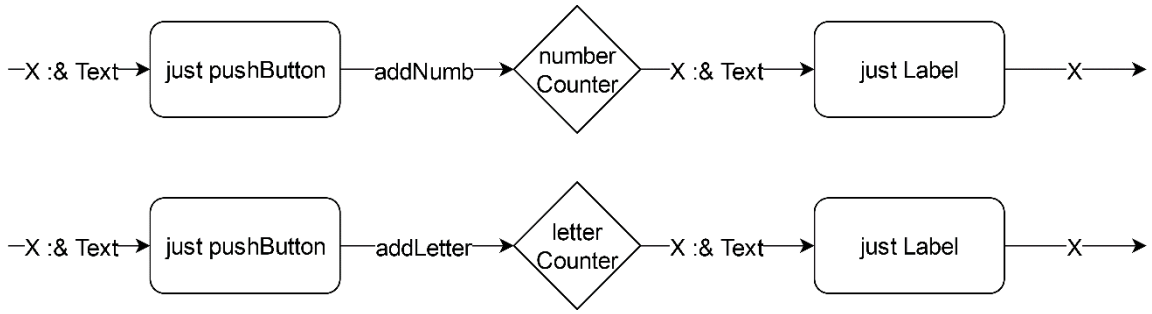
Tagastatakse väärtus *closure*, mis ühtib funktsiooni *mainCircuit* nõutava tagastustüübiga *Closure ::: DSignal 'Of' ()*.



Joonis 16. Funktsiooni mainCircuit skeem.

Kasti sisu tuleb funktsioonist *boxContent*, mille tööd on kujutatud joonisel 17. Funktsioonil *boxContent* on sarnane signatuur funktsiooniga *mainCircuit*, kus sisendiks ja väljundiks nõutakse väärtust *()*. Konstrueeritakse nool, kus sisendi väärtus pole oluline ning väljundiks on väärtus *()*. Funktsiooni sees on kaks loendurit:

- *numberCounter* on arvuline loendur ning me tahame, et lipikul kujutataks nupuvajutuste kordade arv, niisiis on vaja kasutada segmenteeritud signaali, kus algväärtus on 0 ning iga nupuvajutus liidab ühe juurde. Selleks kasutame funktsiooni *SSignal.scan*. Funktsioon *scan* nõuab sisendiks akumulaatori algväärtust, mis on antud juhul 0. Teiseks sisendiks nõuab funktsiooni millega akumulaatorit muuta ning kolmas sisend peab olema diskreetne signaal, mis *scan* tegevust käivitaks. Antud juhul on kolmandaks sisendiks *addNumb* (tuleb *pushButton* kaudu). Väljundiks annab funktsioon *scan* segmenteeritud signaali.
- *letterCounter* on sõne loendur, kus kogu tegevus on väga sarnane *numberCounter* muutujale, kuid siin on akumulaatoriks tühi sõne ja liidetakse juurde tähte 'X'.



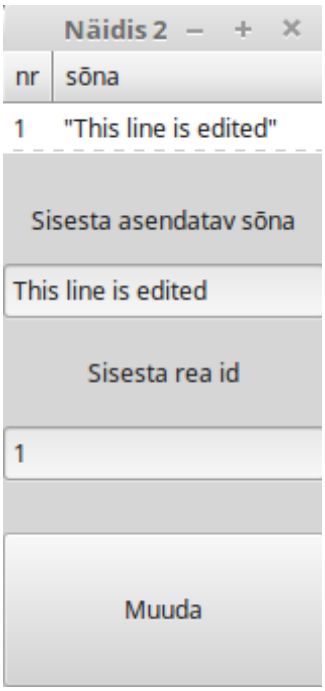
Joonis 17. Funktsiooni boxContent töö.

Luuakse neli kasutajaliidese elementi: kõigepealt luuakse kaks nuppu *just pushButton*, mis nõuavad sisendiks mõlemal juhul *(X :&(Req Text ::: (SSignal 'Of" String)))* tüüpi hoidlaid. Need sobitatakse hoidlate *X :& Text := pure "Suurenda arvu"* ja *X:& Text := pure "Lisa täht"* abil. Väljundiks on mõlema nupu puhul *(X :& (Push ::: (DSignal 'Of" ())))* tüüpi väärtus, mis sobitatakse esimese nupu puhul väärtusega *X :& Push := addNumb* ning teise nupu puhul väärtusega *X :& Push := addLetter*.

Järgmisena on vaja luua kaks silti, et kuvada saadav info. Seda tehakse *just label* toel, mis nõuab sisendiks *(X :& (Req Text ::: (SSignal 'Of" String)))* tüüpi ja see sobitatakse *X :& Text := numberCounter* ja *X :& Text := letterCounter* toel. Väljundiks on mõlema sildi puhul tühi hoidla, mis sobitatakse väärtusega *X*. Tagastatakse väärtus *()*, mis ühtib funktsiooni *boxContent* signatuuriga.

4.2 Example2

Näidis on kirjutatud töö autori poolt ning see näitab loenditega töötamist läbi signaalide. Hulk on juba eelnevalt täidetud elementidega (id, element) ning need on jaotatud veergudesse. Sisestades vastava olemasoleva id lahtrisse ning vajutades nuppu 'Muuda' asendatakse vastav rida väärtusega 'X'. Näiteks joonisel 18 on kujutatud antud näidist, kus hulka on juba muudetud.



Joonis 18. Example2.hs

```
mainCircuit :: (ContainerUIBackend uiBackend) =>
                    UICircuit Window uiBackend era () (DSignal era ())
mainCircuit = proc () -> do
        X :& Closure := closure `With` X `With` ()
                <- window `with` box Vertical `with` content
                -< X :& Title := pure "Näidis 2" `With` X `With` ()
        returnA -< closure

content :: (ContainerUIBackend uiBackend) =>
                                  UICircuit Widget uiBackend era () ()
content = proc () -> do
        rec let


            elems   = ISignal.construct (Seq.fromList elements)
                                        ((toUpdate nr rida push))
            cols = ISignal.const        (Seq.fromList [col1,col2])

            col1 = Column "nr"   (\(n,_) ->
                          TextCellDisplay (show (n)) white) textCell
            col2 = Column "sõna" (\(_,n) ->
                          TextCellDisplay (show (n)) white) textCell

            X :& Selection := sel <- just listView -< X
                                                :& Elements := elems
                                                :& Columns  := cols

            X                     <- just label
                     -< X :& Text := pure "Sisesta asendatav sõna"

            X :& Content := rida  <- just lineEditor -< X

            X                     <- just label
                     -< X :& Text := pure "Sisesta rea id"

            X :& Content := nr    <- just lineEditor -< X
            X :& Push    := push  <- just pushButton
                                  -< X :& Text := pure "Muuda"

        returnA -< ()

white     = RGB (fromFactor 1) (fromFactor 1) (fromFactor 1)

toUpdate :: SSignal era String -> SSignal era String ->
               DSignal era () -> DSignal era (Diff (Seq(Int, String)))
toUpdate v r p = updateWith <$ p <#> v <#> r

elements :: [(Int, String)]
elements  = [(1,"a"), (2,"b"), (3,"c"), (4,"d"), (5,"e"), (6,"f")]


updateWith :: String -> String -> (Diff (Seq (Int, String)))
updateWith n r = case reads n of
    [(n,_)] -> Diff $ Seq.singleton $ Update (n-1)
                                             (Seq.singleton (n, r))
```

Joonis 19. Example2 lähtekood (täispikkuses on leitav lisas III).

See näide kasutab võrreldes eelnevate näidetega tüübipiirajat *ContainerUIBackend*, mis annab vajalikud komponendid *listView* ja *textCel*. Näide, mille lähtekood on joonisel 19, algab funktsiooniga *mainCircuit*, mis signatuuri järgi nõuab sisendiks väärtust *()* ning väljundiks on väärtus *closure*.

Funktsioonis *mainCircuit* luuakse nool, milles kasutatakse hoidlat, kus *window* nõuab sisendiks *X :& (Req Title ::: (SSignal `Of` String))* tüüpi ja see omakorda sobitatakse väärtusega *X :& Title := pure "Teine näidis"*. Väljundiks annab aken tüübi *X:& (Closure ::: (DSignal 'Of' ()))*, mis sobitatakse väärtusega *X :& Closure := closure*.

Järgmisena on kasutatud jällegi hoidlat – põhirolli mängib siin *mainWindow* muutuja all olev *window*, mis nõuab sisendiks *X :& (Req Title ::: (SSignal `Of` String))* tüüpi hoidlat. See omakorda sobitatakse *X :& Title := pure "Esimene näidis"* tüüpi hoidlaga, kus *pure* abil saame sõne tüüpi signaali. Väljundiks annab *window X:& (Closure ::: (DSignal 'Of' ()))* tüübi, mis sobitatakse *X :& Closure := closure* kaudu.

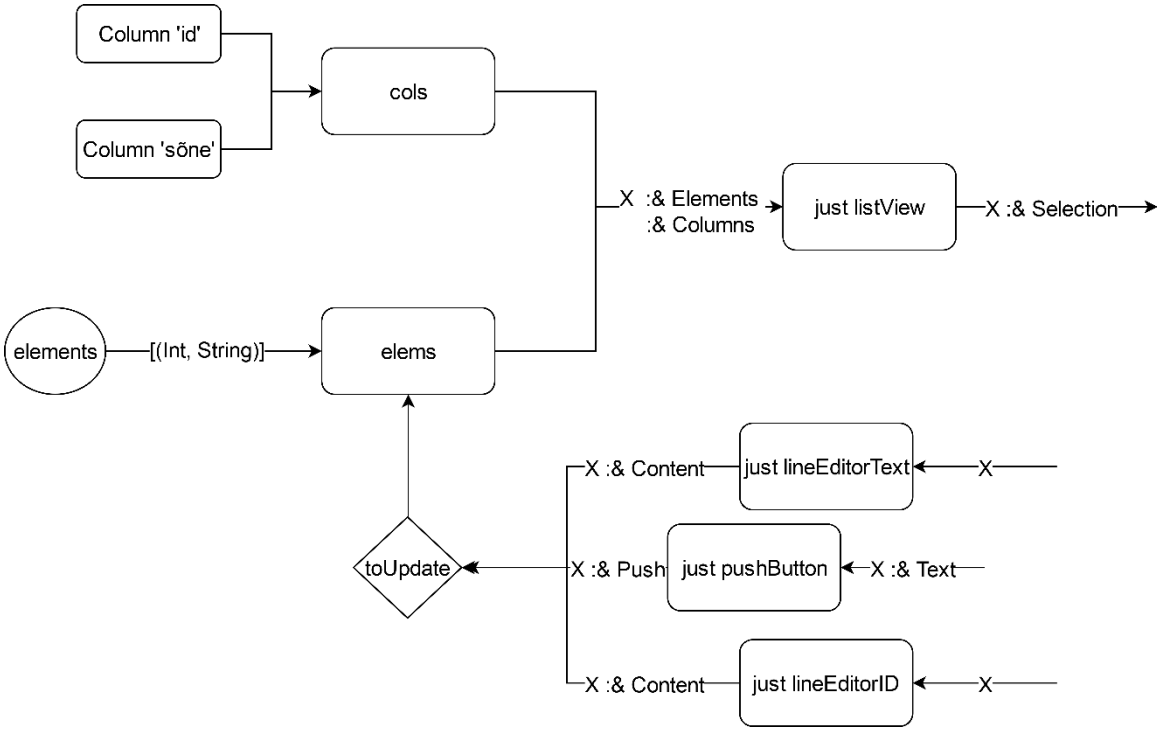
Aknale antakse sisu *'with'* andmekonstruktoriga. Sisuks on *box Vertical*, mis annab väljundiks ja võtab sisendiks tühja hoidla *X*. Hoidlale antakse omakorda sisu *'with'* andmekonstruktoriga ning selleks on *content*, mille üldine struktuur on kujutatud joonisel 20.

Vastavalt joonisele 19 on kahe funktsiooni *content* ja *mainCircuit* signatuurid sarnased. Kuid funktsioon *content* nõuab sisendiks ja väljundiks väärtust *()*. Elemendid loendi *listView* jaoks tulevad muutujast *elems*, mille tüübiks on *X :& (Req Elements ::: (ISignal 'Of' Seq el))*.

Tüübi saamiseks kasutatakse kasvava signaali *construct* meetodit, mis võtab esimeseks sisendiks algseisundi; sisend saadakse loendi algväärtuste nimekirjast *elements*, mis koosneb paaridest (id, väärtus). Loend tehakse tüüpi *Sequence* tänu funktsioonile *Seq.fromList*. Teiseks sisendiks võtab *construct* meetod diskreetse signaali; see signaal tuleb funktsioonist *toUpdater*, millele on sisendiks omakorda *id*, *text* ja *push*:

- Parameeter *id* tuleb *just lineEditor* elemendist, mis saab sisse tühja *Record'i X* ning välja annab X :& Content := id.
- Parameeter *text* tuleb identsest *lineEditor'ist*.
- Parameeter *push* tuleb nupu elemendist *just pushButton*, mis nõuab sisendiks X :& Text := pure "Muuda". Välja annab nupp X :& Push := push.

Funktsioon *toUpdater* käivitab omakorda funktsiooni *updateWith*. Sisendid *id* ja *text*, mis antakse ette *Sampler* klassi parameetriga <#>, tulevad eelnevast funktsioonist *toUpdater*. Funktsiooni *updateWith* mõte on leida hulgast element, mille *id* kattub etteantud sisendiga ning seejärel asendada olemasolev väärtus funktsiooniga *Update*. Selle signatuur on tüüpi *Int -> Seq el -> Diff (Seq el),* kus esimeseks parameetriks on elemendi indeks olemasolevas hulgas ja teine parameeter on uus element, mis on hulga tüüpi.



Joonis 20. Funktsiooni content skeem.

Veerud on salvestatud muutujasse *cols*, mille tüübiks on *X :& (Req Columns ::: (ISignal 'Of' Seq (Column uiBackend el)))*. See väärtustatakse esmalt kasutades kasvava signaali meetodit *const*, millele antakse sisse *Seq* tüüpi hulk. See hulk koosneb kahest *Column* elemendist, elementide tüübiks on *String -> (el -> display) -> Cell uiBackend display -> Column uiBackend el*:

- *col1* on indeksi veerg ja sinna lisatakse *textCell* tüüpi elemente, mis saadakse kui võetakse paarist (id, element) esimene element ning kasutatakse sellel funktsiooni *TextCellDisplay*, mis saab ette ühe *String* tüüpi elemendi.
- *col2* on teksti veerg, mis on identne eelneva veeruga.

Kasutajaliidese element *just listView* nõuab sisendiks väärtust tüüpi *(((X :& (Req Elements ::: (ISignal 'Of' Seq el))) :& (Req Columns ::: (ISignal 'Of' Seq (Column uiBackend el)))) :& (Opt HasScrollbars ::: (SSignal 'Of' (Orientation))))*, see sobitatakse *X :& Elements :=*

elems :& Columns := cols. Väljundiks annab tüüpi *(X :& (Selection ::: (SSignal 'Of' Seq el)))* ja see sobitatakse väärtusega *X :& Selection := sel.*

5. Kokkuvõte

Käesolevas lõputöös uuriti ja näidati programmeerimiskeele Haskell teegi Grapefruit erinevaid võimalusi ja eeliseid. Loodi kaks näiteprogrammi ning selgitati lahti üks varem publitseeritud näitekood, millega näidati teegi tähtsamaid funktsioone ja omadusi.

Töö teises peatükis anti ülevaade, mis on funktsionaalne reaktiivne programmeerimine, millistest osadest see koosneb ja kuidas need osad omavahel käituvad. Seda tehti peamiselt nelja näite abil, millest kaks iseloomustasid sündmust ning ülejäänud käitumist.

Lõputöö kolmandas peatükis selgitati kolme peamist süntaksit, mida kasutati järgnevates näitekoodide seletustes. Nende pikem selgitamine oli vajalik, et lugeja saaks aru näitekoodi seletustest ilma lisa lugemata. Teises peatükis on ka lahti seletatud üks teegi näidis Converter, mis näitab kasutajaliidese kõige elementaarsemate elementide sildi ja tekstiredaktori käitumist vastavalt sisendile.

Neljandas peatükis on lõputöö autori kaks näidiskoodi, millest esimene on lühem ja intuitiivsem ning teine keerulisema ülesehitusega. Esimene näidiskood näitab nuppude ja siltide koostööd ning nupuvajutusel toimuvad uuendused eelnevalt seletatud signaalide toel, kus ühele nupule vajutades suureneb arvuloendur ning teisele nupule vajutades suureneb sõne ühe ’X’ võrra. Teine näidiskood näitab hulkade käsitlemist, kus on hulk elemente, mida on võimalik muuta sisestades ühte lahtrisse muudetava rea id ning teise lahtrisse uue väärtuse.

6. Viidatud kirjandus

[1] Blackheath S., Jones A. Functional Reactive Programming. Greenwich, Connecticut: Manning Publications. 2015.

[2] Jeltsch W. Signals, Not Generators!: Trends in Functional Programming, 2009, 145-160.

[3] Jeltsch W. Generic record combinators with static type checking: PPDP, 2010.

[4] Nestra H. Sissejuhatus funktsionaalsesse programmeerimisse. Tartu: Tartu Ülikooli Kirjastus. 2010.

[5] Paterson R. A new notation for arrows: ACM SIGPLAN Notices. 2001, 229-240.

[6] Functional Reactive Programming https://wiki.haskell.org/Functional_Reactive_Programming (25.04.2017)

[7] Elliot C., Hudak P. Functional Reactive animation: ICFP, 1997. http://conal.net/papers/icfp97/ (28.04.2017)

[8] Jeltsch W. Functional reactive programming: Grapefruit-project https://hub.darcs.net/jeltsch/grapefruit (02.05.2017)

[9] Grapefruit https://wiki.haskell.org/Grapefruit (05.05.2017)

Lisad

I. Näidise Example1 lähtekood

```
{-# LANGUAGE Arrows, Rank2Types, TypeOperators #-}
module Example1 (

  mainCircuit

) where

-- Control
import Control.Applicative as Applicative
import Control.Arrow       as Arrow

-- Data
import Data.Record        as Record
import Graphics.UI.Grapefruit.GTK as GTK

-- FRP.Grapefruit
import FRP.Grapefruit.Signal.Discrete  as DSignal
import FRP.Grapefruit.Signal.Segmented as SSignal

-- Graphics.UI.Grapefruit
import Graphics.UI.Grapefruit.Item          as UIItem
import Graphics.UI.Grapefruit.Circuit       as UICircuit
import Graphics.UI.Grapefruit.Backend.Basic as BasicUIBackend

-- |The circuit describing the whole application.

mainCircuit :: (BasicUIBackend uiBackend) =>
                       UICircuit Window uiBackend era () (DSignal era ())
mainCircuit = proc _ -> do
        X :& Closure := closure `With` X `With` _  <- mainWindow
                    -< X :& Title := pure "Esimene naidis" `With` X `With` ()
        returnA -< closure where

        mainWindow  =  window `with` BasicUIBackend.box Vertical
                                                    `with` boxContent

boxContent :: (BasicUIBackend uiBackend) =>
                                  UICircuit Widget uiBackend era () ()
boxContent = proc _ -> do
    rec let
            numberCounter = show <$> SSignal.scan 0
                                          (\xs () -> (1 + xs)) addNumb
            letterCounter = SSignal.scan ""
                                          (\xs () -> ('X' :xs)) addLetter

        X :& Push := addNumb      <- just pushButton
                                     -< X :& Text := pure "Suurenda arvu"
        X :& Push := addLetter    <- just pushButton
                                     -< X :& Text := pure "Lisa täht"
        X <- just label -< X :& Text := numberCounter
        X <- just label -< X :& Text := letterCounter

        returnA -< ()
```

II. Näidise Example2 lähtekood

```
{-# LANGUAGE Arrows, Rank2Types, TypeOperators #-}
module Example2 (

  mainCircuit

) where

-- Control
import Control.Applicative as Applicative
import Control.Arrow       as Arrow

-- Data
import Data.Record        as Record

import Data.Sequence        as Seq
import Data.Colour.RGBSpace as RGBSpace

import Data.Fraction        as Fraction

-- FRP.Grapefruit
import FRP.Grapefruit.Signal
import FRP.Grapefruit.Signal.Discrete    as DSignal
import FRP.Grapefruit.Signal.Segmented   as SSignal
import FRP.Grapefruit.Signal.Incremental as ISignal
import FRP.Grapefruit.Signal.Incremental.Sequence

-- Graphics.UI.Grapefruit
import Graphics.UI.Grapefruit.Item          as UIItem hiding (box)
import Graphics.UI.Grapefruit.Circuit       as UICircuit
import Graphics.UI.Grapefruit.Backend.Basic as BasicUIBackend
import Graphics.UI.Grapefruit.Backend.Container as ContainerUIBackend
import Graphics.UI.Grapefruit.GTK

-- |The circuit describing the whole application.

mainCircuit :: (ContainerUIBackend uiBackend) =>
                  UICircuit Window uiBackend era () (DSignal era ())
mainCircuit = proc () -> do
        X :& Closure := closure `With` X `With` ()
                <- window `with` box Vertical `with` content
                -< X :& Title := pure "Näidis 2" `With` X `With` ()

        returnA -< closure

content :: (ContainerUIBackend uiBackend) =>
                          UICircuit Widget uiBackend era () ()
content = proc () -> do
        rec let

            elems   = ISignal.construct (Seq.fromList elements)
                                        ((toUpdate nr rida push))
            cols = ISignal.const     (Seq.fromList [col1,col2])

            col1 = Column "nr"   (\(n,_) ->
                          TextCellDisplay (show (n)) white) textCell
```

```
            col2 = Column "sõna" (\(_,n) ->
                           TextCellDisplay (show (n)) white) textCell

            X :& Selection := sel <- just listView -< X :& Elements := elems
                                                    :& Columns  := cols

            X                    <- just label
                              -< X :& Text := pure "Sisesta asendatav sõna"

            X :& Content := rida  <- just lineEditor -< X
            X                    <- just label
                              -< X :& Text := pure "Sisesta rea id"

            X :& Content := nr    <- just lineEditor -< X
            X :& Push    := push  <- just pushButton
                              -< X :& Text := pure "Muuda"

         returnA -< ()

white     = RGB (fromFactor 1) (fromFactor 1) (fromFactor 1)

toUpdate :: SSignal era [Char] -> SSignal era [Char] ->
               DSignal era () -> DSignal era (Diff (Seq(Int, String)))
toUpdate v r p = updateWith <$ p <#> v <#> r

elements :: [(Int, String)]
elements  = [(1,"a"), (2,"b"), (3,"c"), (4,"d"), (5,"e"), (6,"f")]


updateWith :: String -> String -> (Diff (Seq (Int, String)))
updateWith n r = case reads n of
    [(n,_)] -> Diff $ Seq.singleton $ Update (n-1) (Seq.singleton (n, r))
```

III. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Karl-Martin Uiga**,
(*autori nimi*)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
Tüübiohutu FRP teegi uurimine: Grapefruit,
(*lõputöö pealkiri*)

mille juhendaja on Kalmer Apinis,
(*juhendaja nimi*)

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **11.05.2017**