TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Tenno Veber

Haskelli FRP teegi uurimine: Reactive-banana

Bakalaureusetöö (9 EAP)

Juhendaja(d):
Kalmer Apinis

Tartu 2017

Haskelli FRP teegi uurimine: Reactive-banana

Lühikokkuvõte:

Käesolevas lõputöös tutvustatakse funktsionaalset reaktiivset programmeerimist, *reactive-banana* teeki ning selgitatakse nelja näidisprogrammi. Nende näidisprogrammidega näidatakse *reactive-banana* teegi kõige tähtsamaid funktsioone, samuti mitmeid viise sündmuste ja käitumiste kasutamiseks ja kombineerimiseks.

Lõputöö põhjal saab lugeja anda hinnangu, kui kasutajasõbralik ja kasulik on FRP teek *reactive-banana*.

Võtmesõnad:

Funktsionaalne reaktiivne programmeerimine, reactive-banana, Haskell

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

Research of Haskell's FRP library: Reactive-banana

Abstract: This Bachelor's thesis explains functional reactive programming, reactive-banana library and four different example programs. Reactive-banana library's most important functions, events and behaviours are shown and combined in example programs.

The reader of this Bachelor's thesis can evaluate user-friendliness and usefulness of FRP library called reactive-banana.

Keywords:

Functional Reactive Programming, reactive-banana, Haskell

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

Sisukord

1. Sissejuhatus

Aastal 1997 avalikustati autorite Conal Elliott'i ja Paul Hudak'u poolt *Functional Reactive Animation* [1] ehk *FRAN*, mis oli autorite sõnul mõeldud interaktiivsete animatsioonide tegemiseks, kuigi kirjeldatud abstraktsioon oli palju üldisemalt kasutatav. *FRAN*-ist on edasi arenenud funktsionaalne reaktiivne programmeerimine.

Stephen Blackheath'i ja Anthony Jones'i raamatus [2] on välja toodud, et suurte programmide arendamisel tuleb tihti ette olukordi, kus ühel hetkel sujub arendus hästi, kuid järgmisel hetkel on arendus jäänud seisma ning iga järgmine arenduse samm võtab oodatust rohkem aega. Selline olukord võib tekkida liiga keeruka koodi tõttu, mis tihtipeale tekib kasutajaliidese põhiste programmide arendamisel. Keeruka koodiga tekkivaid probleeme saab vältida mitmel viisil ning üheks viisiks on funktsionaalse reaktiivse programmeerimise ehk FRP rakendamine juba projekti algfaasides. Funktsionaalne reaktiivne programmeerimine teeb koodi paremini loetavamaks, mis omakorda teeb hooldamise ja arendamise kiiremaks ning lihtsamaks.

Käesolev lõputöö uurib funktsionaalse reaktiivse programmeerimise teeki nimega *reactive-banana* [3], mille autoriks on Heinrich Apfelmus.

Lõputöö eesmärgiks on anda lugejale ülevaade funktsionaalsest reaktiivsest programmeerimisest, tutvustada *reactive-banana* teegi kõige olulisemaid võimalusi nii näidisprogrammide kui ka teksti abil. Lõputöö põhjal saab lugeja anda hinnangu kui kasutajasõbralik ja kasulik on FRP peale kahekümne aastast teooria arendamist.

Lõputöö esimene peatükk on sissejuhatus. Teine peatükk tutvustab FRP-d, sündmusi ja käitumisi. Kolmas peatükk tutvustab *reactive-banana* teeki ja selgitab nelja näidisprogrammi. Neljas peatükk on kokkuvõte lõputööst. Viies peatükk sisaldab kasutatud materjale. Kuues peatükk ehk lisad sisaldavad nelja näidisprogrammi koodi ning litsentsi.

2. Funktsionaalse reaktiivse programmeerimise tutvustus

Järgnevas peatükis tutvustatakse funktsionaalset reaktiivset programmeerimist, sündmusi ja käitumisi.

2.1 Funktsionaalse reaktiivse programmeerimise ülevaade

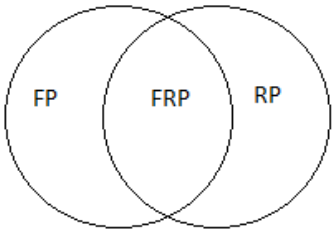
Alljärgnev alampeatükk on refereeritud Stephen Blackheath'i ja Anthony Jones'i raamatust [2], kus selgitatakse, et mida suurem on projekt, seda keerulisem on seda hallata. Selline probleem tekib liiga keeruka koodi tõttu, mida saab ära hoida kasutades funktsionaalset reaktiivset programmeerimist.

Funktsionaalset reaktiivset programmeerimist ehk FRP-d, mis tuleb lihtsa teegina, on kerge juba olemasoleva projekti külge ühendada. FRP osa projektis võib olla väga väike, kuid ka terve projekt võib koosneda FRP-st.

Funktsionaalne programmeerimine on programmeerimisstiil, kus ei ole muutujaid ja kõik tegevused toimuvad funktsioonidega [2,4]. Funktsionaalses programmeerimises kirjeldatakse iga sisendi jaoks väljundit ning kuna puuduvad muutujad, ei saa teha kuulareid.

Reaktiivne programmeerimine tähendab, et programm on sündmustepõhine ja reageerib sisenditele [2,5]. Näiteks kuularitega programmeerimine, kus kuular jälgib nupuga seotud tegevusi ja nupuvajutuse korral reageerib sellele.

Funktsionaalne reaktiivne programmeerimine ehk FRP koosneb funktsionaalsest ja reaktiivsest programmeerimisest, täpsemalt on FRP nende kombineerimine. Funksionaalse programmeerimise, reaktiivse programmeerimise ja FRP sõltuvust iseloomustab joonis 1.



Joonis 1. FRP on FP ja RP alamhulk [2].

Funktsionaalne reaktiivne programmeerimine pakub elegantset viisi interaktiivsete programmide loomiseks, seejuures välditakse segast koodi, mis tihtipeale tekib kasutajaliideste programmeerimisel. Lisaks koodi puhtamaks tegemisele, on kood ka

lihtsalt hooldatav ning loetav. FRP on hea lahendus kuularite (ingl. k. *listener*) ja tagasikutsetega (ingl. k. *callback*) tekkivate probleemide lahendamiseks. Enamlevinud kuularite ja tagasikutsete probleemideks on ootamatu järjekord, märkamata jäänud sündmuste toimumine, lekkivad tagasikutsed ja tahtmatu rekursiooni esinemine.

Funktsionaalse reaktiivse programmeerimise olemus peitub käitumises, mille põhjustab ajakulg. FRP integreerib aja, aja muutumise ja sündmused funktsionaalseks programmeerimiseks [6]. See võimaldab teha graafilisi kasutajaliideseid, animatsioone, roboteid jms.

FRP on ajas muutuva väärtuse kirjeldamine teatud ajahetkel, mida võib kirjeldada ka kui reaktsiooni programmi sisenditele [7]. Näiteks funktsioon, mille algväärtus on 0 ning iga nupuvajutuse korral liidetakse väärtusele arv 1. Sisendiks oleks nupuvajutus ning reaktsiooniks arvu 1 liitmine väärtusele.

Lisaks on Stephen Blackheath ja Anthony Jones [2] veel kirjutanud, et FRP sunnib programmeerijat teisiti mõtlema. Programmi tuleks vaadata kui „mis see on“, mitte „mida see teeb“. FRP ei kasuta muutuvate väärtustega muutujaid, vaid kirjeldab programmi ajas muutuvate funktsioonidega ehk *Behavior*'itega [7].

FRP kasutab kahte põhilist andmetüüpi: väärtusi, mis muutuvad ajas, ehk *Behavior*'e ning sündmuste vooge ehk *Event*'e [3].

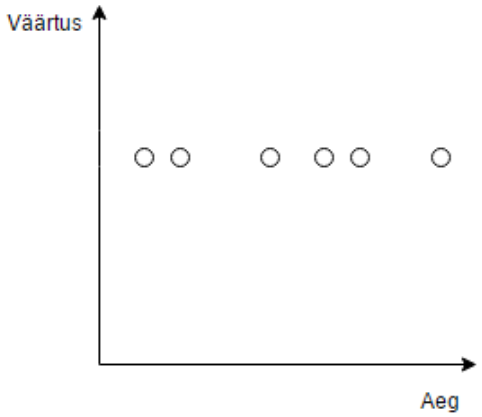
2.2 Event ehk sündmus

Alljärgnev alampeatükk tugineb artiklitele [3,8], kus selgitatakse, et *Event a* väljendab mitmete sündmuste toimumist läbi aja. *Event a* ei tähenda kunagi ühte kindlat sündmust, vaid alati sündmuste voogu. Andmetüüp „*Event a“* tähistab sündmuste voo tüüpi, mille väärtused on tüübist *a*. Näiteks sõne tüüpi sündmuste tüübiks on „*Event String*“.

Event on defineeritud järgnevalt: *type Event a = [(Time,a)]*. Lühidalt öeldes on tegemist listiga, mis sisaldab paare *(Time, a)*, kus paari esimene komponent tähistab sündmuse toimumise aega ja teine komponent sündmuse väärtust. Igat paari kutsutakse sündmuse toimumiseks (ingl. k. *event occurrence*). Ühes sündmuse voos ei saa samal ajal toimuda kahte sündmust.

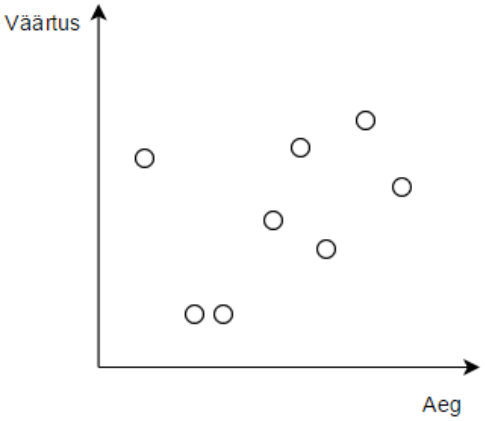
Sündmuseks võib olla kursoriga programmi nupule vajutus. Sellise sündmuse puhul ei ole tähtis sündmuse väärtus, vaid oluline on see, et sündmus on toimunud. Sündmuse tüübiks

on „*Event ()*“. Joonisel 2 on näha, kuidas aja möödudes toimuvad sündmused, sealjuures sündmuste väärtused on samad.



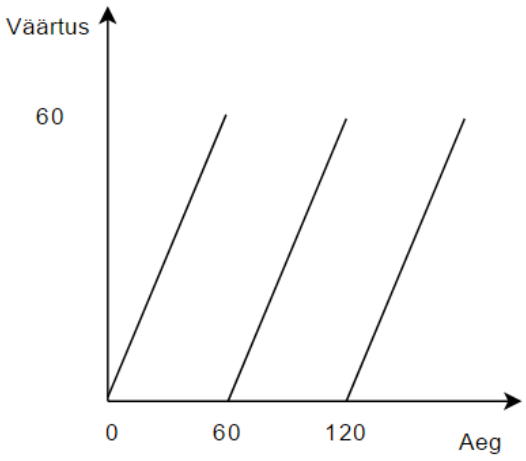
Joonis 2. Nupuvajutuste voog

Sündmuseks võib olla ka klaviatuuri klahvivajutus. Üldiselt selliste sündmuste puhul on väärtus tähtis, kuna on vaja teada, mis nupule vajutati. Sündmuse tüübiks on „*Event Char*“. Joonisel 3 on näha, kuidas aja möödudes toimuvad erinevate väärtustega sündmused.



Joonis 3. Klaviatuuri klahvivajutuste voog

Sündmused võivad olla ka keerulisemad kui klaviatuuri klahvivajutus, näiteks sekundite möödumine. Sekund on sündmus ja sekundid kokku moodustavad sündmuste voo. Sündmuse tüübiks on „*Event Int*“. Joonisel 4 on kujutatud sekundite väärtused sõltuvalt ajast.

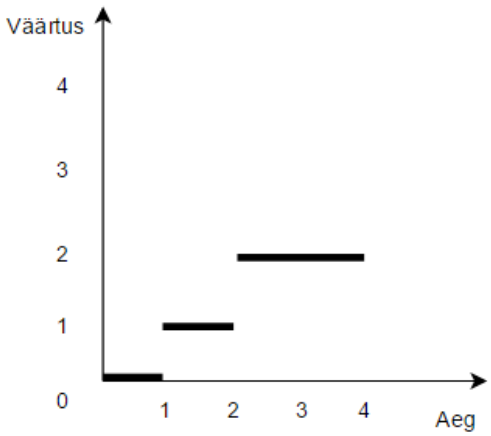


Joonis 4. Sekundite voog

2.3 Behavior ehk käitumine

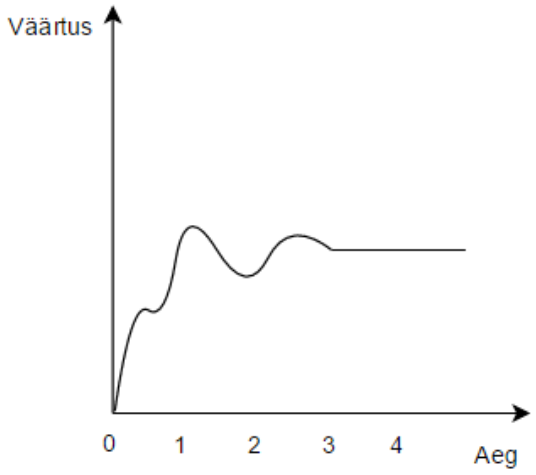
Järgnev lõik on refereeritud artiklitest [3,8], kus selgitatakse, et *Behavior a* on väärtus, mis muutub ajas, kuid igal ajahetkel omab väärtust. Andmetüüp „*Behavior a*“ tähendab käitumist, mille väärtused on tüübist *a*.

Näiteks võib käitumine reageerida nupuvajutusele. Antud näite käitumise algväärtus on 0 ja iga sündmuse ehk nupuvajutuse korral liidetakse väärtusele arv 1. Esimesel ajaühikul vajutatakse nuppu ühe korra ning käitumise väärtus muutub üheks. Teisel ajaühikul vajutatakse nuppu veel ühe korra ning seetõttu liidetakse käitumise väärtusele arv 1, seega on käitumise väärtus kaks. Kuna kolmandal ajaühikul ei vajutata nuppu, siis käitumise väärtus jääb samaks, mis teisel ajaühikul. Antud näidet iseloomustab joonis 5.



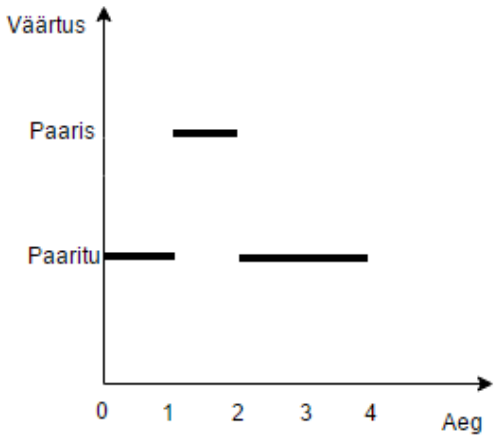
Joonis 5. Nupuvajutuse käitumine

Käitumine võib olla ka hiire sündmuse põhjal, näiteks hiire liigutamine. Hiire liigutamine toob kaasa väärtuse muutumise. Joonisel 6 on näha, et esimesel kolmel ajaühikul liigutatakse hiirt ning järgnevatel ajaühikutel hiirt ei liigutada, seega käitumise väärtus ei muutu.



Joonis 6. Kursori liikumise käitumine

Väga sageli kasutatakse ka käitumisi, mis reageerivad tekstiväljale sisestatud sõnele. Näiteks võib käitumine reageerida tekstiväljale sisestatud arvule, olenevalt sellest, kas sisestatud arv on paaris või paaritu arv. Kuna algväärtuseks on arv 1, siis käitumise väärtus on „Paaritu“. Ajahetkel 1 sisestakse tekstiväljale arv 4, selle tõttu käitumise väärtuseks saab „Paaris“. Ajahetkel 2 sisestatakse tekstiväljale arv 3, seega käitumise väärtus muutub „Paaritu“-ks ning kuna tekstivälja enam ei muudeta, siis käitumise väärtus jääb kuni programmi töö lõpuni „Paaritu“. Sellist käitumist iseloomustab joonis 7.



Joonis 7. Tekstiväljale sisestatud arvude käitumine

3. Reactive-banana

Järgnevas peatükis tutvustatakse *reactive-banana* teeki ja selgitatakse nelja näidisprogrammi.

3.1 Reactive-banana tutvustus

Reactive-banana [3] on funktsionaalse reaktiivse programmeerimise teek, mille autoriks on Heinrich Apfelmus. *Reactive-banana* avalikustati esmakordselt 2011. aasta märtsis, viimane versioon 1.1.0.0 aga 2016. aasta jaanuaris. Kokku on avalikustatud 11 *reactive-banana* versiooni. Inimestele, kes esmakordselt puutuvad kokku *reactive-banana* teegiga, on abiks mitmed avalikud näidisprogrammid [9], mis aitavad alustada teegi kasutamist.

Teegi autori sõnul [7,10] on *reactive-banana* loodud kasutamaks koos teiste teekidega. Näiteks sobib *reactive-banana* hästi kokku sündmusepõhiste kasutajaliideste raamistikega.

Mõistmaks *reactive-banana*'t, tuleb aru saada FRP olemusest. FRP tuumaks on tüüp *Event* ehk sündmus, *Behavior* ehk käitumine ning mitmed viisid nende kahe tüübi kombineerimiseks [3].

Funktsionaalset reaktiivset programmeerimist seletavas artiklis [8] selgitatakse, et sündmuste voogudega saab teha kolme põhilist tegevust: muuta sündmusi voogudes, filtreerida sündmusi ning kombineerida sündmuste vooge. Sündmused ja käitumised asuvad sündmuste võrgustikus (ingl. k. *event network*) ning väljaspool võrgustikku ei ole neid võimalik kasutada.

Näidisprogrammides protseduuride loomisel kasutatakse *do* süntaksit, mis on loodud monaadilise arvutamise intuitiivsemaks tegemiseks [4]. *Do* süntaksiga saab luua *IO* tüüpi lausete ploki, kus lauseid täidetakse üksteise järel. Allolev koodijupp trükib välja „a" ning seejärel „b".

```
do  putStrLn "a"
    putStrLn "b"
```

Lisaks kasutatakse ka <- süntaksit, mis võimaldab väärtuste salvestamist muutujatesse [4]. Allolev koodijupp loeb funktsiooni *getLine :: IO String* abil terminalist kasutaja poolt sisestatud teksti, salvestab teksti muutujasse *x :: String* ning trükib selle välja funktsiooni *putStrLn :: String -> IO String* abil.

```
do  x <- getLine
    putStrLn x
```

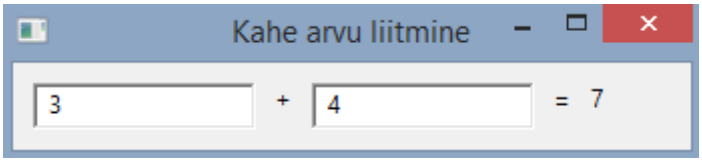
Reactive-banana funktsioonide tüübid ja selgitused on võetud *API*-st [3]. Näiteks funktsiooniga *stepper :: monadMoment m => a -> Event a -> m (Behavior a)* on sündmusi võimalik teisendada käitumisteks. Rakendades seda funktsiooni klaviatuuri klahvivajutuse sündmusele saadakse käitumine, mis väljendab viimast klahvivajutust.

Antud näites on funktsiooni *stepper* esimeseks argumendiks käitumise *k* algväärtus ehk tühi sõne ning iga kord kui kasutaja vajutab klaviatuuri klahvi ehk toimub sündmus *s*, asendatakse käitumise väärtus vajutatud tähe sümboliga.

```
k <- stepper "" s
```

3.2 Kahe arvu liitmine

Antud näidisprogramm on tehtud avaliku näite „Arithmetic.hs“ [9] põhjal. Programmi eesmärgiks on kahe sisestatud arvu liitmine ning summa väljastamine. Näidisprogrammi kood asub lisades (Lisa 1). Joonisel 8 on näidatud antud näidisprogrammi kasutajaliides.



Joonis 8. Näidisprogrammi 1 kasutajaliides

Alustuseks loome meetodi *gui*, mille tüübiks on *IO ()*.

```
gui :: IO ()
```

Meetod *gui* algab aknaraami loomisega, mille nimeks on *f* ning tekstiks on „Kahe arvu liitmine“. Kuna programmi eesmärk on kahe sisestatud arvu liitmine ning nende summa väljastamine, on vaja kahte sisestusvälja ja ühte staatilist tekstivälja. Nende muutuja-nimedeks on vastavalt *input1*, *input2* ja *output*.

Sisestusväljade loomiseks kasutatakse *entry* funktsiooni, mis võtab sisse kaks argumenti. Esimeseks argumendiks on aknaraam *f* ja teiseks argumendiks on list, mis sisaldab tekstivälja andmeid, näiteks teksti.

Staatilise tekstivälja loomisel kasutatakse funktsiooni *staticText*, mis võtab sarnaselt sisestusväljale kaks argumenti.

```
gui = do
    f         <- frame [text := "Kahe arvu liitmine"]
    input1    <- entry f []
    input2    <- entry f []
    output    <- staticText f []
```

Aknaraami *f* elementide paigutamiseks kasutatakse funktsiooni *set*. Aknaraamistikule paigutatakse elemendid *input1*, *input2*, *output* ning sildid „+“ ja „=“ [9].

```
    set f [layout := margin 10 $ row 10
            [widget input1, label "+", widget input2
            , label "=", minsize (sz 40 20) $ widget output]]
```

Sündmuste ja käitumiste loomiseks luuakse sündmuste võrgustik, nimega *networkDescription*, mille tüüp on *MomentIO ()*. *MomentIO* on monaad, kus saab teha sisend-väljund operatsioone nagu *IO* monaadis, kuid lisaks saab kasutada sündmuste võrgustike loomise funktsioone.

```
    let networkDescription :: MomentIO ()
        networkDescription = do
```

Seejärel luuakse muutujad *binput1* ja *binput2*, mis saavad oma väärtused funktsiooni *behaviorText* abil [9]. *BehaviorText* on funktsioon, mille argumentideks on tekstiväli ehk eelnevalt mainitud *input1* või *input2* ning algväärtus ehk antud näite puhul tühi sõne. Tagastustüüp on *MomentIO (Behavior String)*.

```
            binput1  <- behaviorText input1 ""
            binput2  <- behaviorText input2 ""
```

Selles näites on kaks sisendit *binput1 :: Behavior String* ja *binput2 :: Behavior String*, mis võimalusel teisendatakse numbrilisele kujule. Selleks kasutatakse puhast funktsiooni *readNumber* [9]. *ReadNumber* väärtuseks on tühja listi puhul *Nothing* ning täisarvulise sisendi puhul *Just a*, kus *a* tähistab arvu.

```
            let
                readNumber :: String -> Maybe Int
                readNumber s = listToMaybe [x | (x,"") <- reads s]
```

Operaator <*$*> on infix sünonüüm funktsioonile *fmap :: (a -> b) -> Behavior a -> Behavior b*, mis tuleneb tüübiklassist *Functor*. Nimelt võtab funktsioon *fmap* sisse kaks argumenti – esimeseks argumendiks on funktsioon, mis omakorda saab sisse *a* tüüpi

väärtuse ning tagastab *b* tüüpi väärtuse, ning teiseks argumendiks *Behavior a*. Tagastatakse *Behavior b*, mis on saadud *Behavior a*'st, millele on rakendatud funktsiooni *(a -> b)*.

Järgmisena luuakse käitumised *bmayint1* ja *bmayint2*, mille tüüpideks on *Behavior (Maybe Int)*. Käitumiste eesmärkideks on sisestusväljale sisestatud arvu teisendamine numbrilisele kujule. Selleks kasutatakse üleval mainitud funktsiooni *readNumber* ja *fmap*.

```
bmayint1,bmayint2 :: Behavior (Maybe Int)
bmayint1 = readNumber <$> binput1
bmayint2 = readNumber <$> binput2
```

Kahe arvu liitmiseks kasutatakse puhast funktsiooni *addNumber :: Maybe Int -> Maybe Int -> Maybe Int*, mis kahe täisarvulise sisendi korral tagastab nende summa, vastasel korral *Nothing*.

```
addNumber :: Maybe Int -> Maybe Int -> Maybe Int
addNumber (Just a) (Just b) = Just (a+b)
addNumber _ _ = Nothing
```

Järgmiseks luuakse käitumine, mille nimeks on *bsum* ning tüübiks *Behavior (Maybe Int)*. Kombinaator *<*> :: Behavior (a -> b) -> Behavior a -> Behavior b* rakendab ajas muutuvat funktsiooni ajas muutuvale väärtusele. Kombinaator <*> tuleneb tüübiklassist *Applicative* ja seda kasutatakse tihti koos operaatoriga *<$>*.

Käitumine *bsum* kasutab üleval mainitud funktsioone *addNumber* ja *fmap* ning kombinaatorit <*>, mille abil liidetakse sisendväljadele sisestatud täisarvulised arvud.

```
bsum :: Behavior (Maybe Int)
bsum = addNumber <$> bmayint1 <*> bmayint2
```

Kuna arvude summa on *Maybe Int* tüüpi ning summa kuvamiseks on vaja *String* tüüpi, kasutatakse funktsiooni *showNumber :: Maybe Int -> String*, mis *Just a* tüüpi sisendi puhul saab väärtuseks *show a*, vastasel korral „--" [9].

```
showNumber :: Maybe Int -> String
showNumber (Just a) = show a
showNumber Nothing = "--"
```

Kahe sisestatud arvu summa *String* tüüpi tegemiseks luuakse käitumine, mille nimeks on *result*. Käitumine kasutab funktsioone *showNumber* ja *fmap*, mille abil *bsum* väärtused muudetakse *String* tüüpi väärtuseks.

```
result :: Behavior String
result = showNumber <$> bsum
```

Arvutatud summa ühendatakse kasutajaliidesega *sink :: w -> [Prop' w] -> MomentIO ()* funktsiooni abil, kus esimene argument on staatiline tekstiväli *output* ning teine argument on list, mis sisaldab teksti, mille väärtus on käitumine *result*.

```
sink output [text :== result]
```

Järgmisena kompileeritakse võrgustik *networkDescription* ning käivitatakse funktsiooni *actuate* abil.

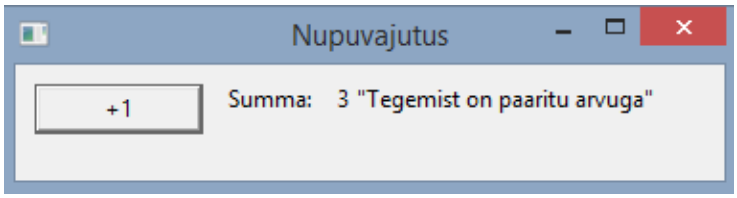
```
network <- compile networkDescription
actuate network
```

Viimasena loome meetodi *main*, mille tüübiks on *IO ()*. *Main* meetodis kasutatakse funktsiooni *start*, mis käivitab kasutajaliidese *gui*.

```
main :: IO ()
main = start gui
```

3.3 Nupuvajutus

Eesmärgiks on teha programm, mis loeb mitu korda on vajutatud nuppu ning ütleb, kas nupuvajutuste summa on paarisarv või paaritu arv. Näidisprogrammi kood asub lisades (Lisa 2). Joonisel 9 on näidatud antud näidisprogrammi kasutajaliides.



Joonis 9. Näidisprogrammi 2 kasutajaliides.

Selleks loome kõigepealt *gui* meetodi, mis on *IO ()* tüüpi. Seejärel loome aknaraami, mille muutujanimeks on *f* ja pealkirjaks „Nupuvajutus". Esmalt lisame aknaraami nupu, muutujanimega *nupp* ning tekstiga „+1". Selleks, et kuvada nupuvajutuste summat ja paaris või paaritut olekut, on loodud kaks staatistilist tekstivälja, vastavalt muutujanimedega *valjund* ja *valjund2*.

```
gui :: IO ()
gui = do
```

```
f         <- frame [text := "Nupuvajutus"]
nupp      <- button f [text := "+1"]
valjund   <- staticText f []
valjund2  <- staticText f []
```

Aknaraami *f* elementide paigutamiseks kasutatakse funktsiooni *set*. Aknaraamistikule paigutatakse elemendid *nupp*, *valjund*, *valjund2* ja silt *„ Summa: “*.

```
set f [layout := minsize (sz 350 40) $ margin 10 $ row 10
    [widget nupp, label " Summa: ", widget valjund
    , widget valjund2]]
```

Järgmisena luuakse sündmuste võrgustik, kus omakorda luuakse sündmused ja käitumised. Võrgustiku loomiseks kasutatakse *let* ja tüübiks määratakse *MomentIO ()*.

```
let networkDescription :: MomentIO ()
    networkDescription = do
```

Võrgustikku luuakse nupuvajutuse sündmus, mille nimeks on *nuppE*. Sündmuse loomisel kasutatakse funktsiooni *event0 :: w -> Event w (IO ()) -> MomentIO (Event ())*. Antud näites on argumentideks *nupp* ehk aknaraami element ning *command*, mis on sündmus *WX* teegis.

```
        nuppE   <- event0 nupp command
```

Järgmiseks luuakse käitumine, mis reageerib nupuvajutusele. Eelnevalt mainitud käitumise nimeks on *behavior1* ja tüübiks *Behavior Int*.

```
        (behavior1 :: Behavior Int) <- accumB 0 ((+1) <$ nuppE)
```

Käitumise väärtuse andmiseks kasutatakse funktsiooni *accumB :: MonadMoment m => a -> Event(a -> a) -> m (Behavior a)*, mis võtab sisse esimese argumendina algväärtuse, milleks antud näites on arv 0, ning teise argumendina funktsiooni, mille tüübiks on *Event*. Antud näidisprogrammi *accumB* teiseks argumendiks määratakse *((+1) <$ nuppE)*, mis on sündmuste voog, kus sündmused toimuvad *nuppE* sündmustega samaaegselt, kuid sündmuse väärtus on asendatud funktsiooniga (+1) ehk liidetakse väärtustele arv 1.

Tulemuseks on käitumine *behavior1*, mis alustades täisarvust 0, iga *nuppE* sündmuse korral suureneb ühe võrra.

```
                                    accumB 0 ((+1) <$ nuppE)
```

Lisaks luuakse käitumine, nimega *behavior2* ning tüübiga *Behavior String*, paaris või paaritu arvu oleku vaatlemiseks. Käitumise väärtuse andmiseks kasutatakse funktsiooni *fmap :: (a -> b) -> Behavior a -> Behavior b*. Antud näites funktsioon *fmap,* mille sünonüüm on *<$>*, võtab esimese argumendina sisse funktsiooni *func* ja teise argumendina eelnevalt mainitud *behavior1*'e.

Funktsioon *func* võtab argumendi *x* ning tagastab „*Tegemist on paarisarvuga"*, kui *x* on paarisarv, või „*Tegemist on paaritu arvuga"*, kui *x* on paaritu arv.

```
behavior2 :: Behavior String
behavior2 = func <$> behavior1
    where
    func x = if even x
             then "Tegemist on paarisarvuga"
             else "Tegemist on paaritu arvuga"
```

Käitumiste *behavior1* ja *behavior2* tulemused ühendatakse kasutajaliidesega funktsiooni *sink :: w -> [Prop' w] -> MomentIO ()* abil.

Antud näites funktsiooni *sink* esimeseks argumendiks on staatiline väli *valjund* või *valjund2*, kuhu väljund kuvatakse, ning teiseks argumendiks on list, mis sisaldab teksti. Tekst saadakse kasutades eelnevalt mainitud funktsiooni *fmap :: (a -> b) -> Behavior a -> Behavior b,* mille esimeseks argumendiks on funktsioon *show* ja teiseks argumendiks *behavior1* või *behavior2.*

```
sink valjund [text :==  show <$> behavior1 ]
sink valjund2 [text :==  show <$> behavior2 ]
```

Kui sündmuste võrgustik on loodud, kompileeritakse võrgustik *networkDescription* ning käivitatakse funktsiooni *actuate* abil.

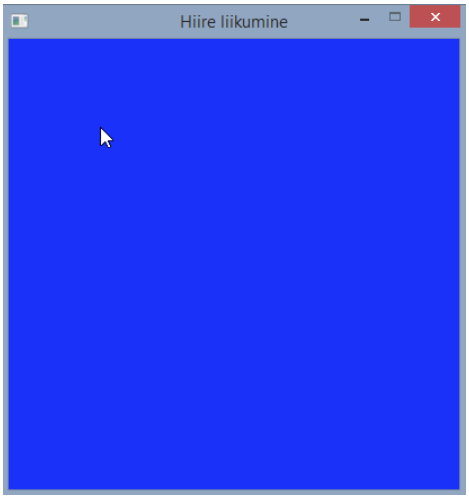
```
network <- compile networkDescription
actuate network
```

Programmi käivitamiseks luuakse meetod *main* ning kasutatakse funktsiooni *start*, mis käivitab kasutajaliidese.

```
main :: IO ()
main = start gui
```

3.4 Kursori asukoha määramine

Antud näidisprogramm on tehtud avaliku näite „Animation.hs“ [9] abil. Programmi eesmärgiks on jälgida kursori asukohta ning vastavalt kursori positsioonile muuta kasutajaliidese taustavärvi. Näidisprogrammi kood asub lisades (Lisa 3). Joonisel 10 on näidatud näidisprogrammi kasutajaliides.



Joonis 10. Näidisprogrammi 3 kasutajaliides.

Esmalt luuakse meetod *gui*, mis on *IO ()* tüüpi. Meetodisse *gui* lisatakse aknaraamistik *f*, mille suurus ei ole muudetav.

Luuakse ka taimer *t*, kasutades funktsiooni *timer :: Window a -> [Prop Timer] -> IO Timer*, mis saab esimeseks argumendiks aknaraamistiku *f* ja teises argumendis määratakse intervalliks 10ms.

Lisaks luuakse paneel *p*, kasutades funktsiooni *panel* :: *Window a -> [Prop (panel ())] -> IO (Panel ())*, mille pealt saab kursori asukohta teada.

```
gui :: IO ()
gui = do
    f  <- frame [ text := "Hiire liikumine", resizeable := False ]
    t  <- timer f [ interval := 10 ]
    p  <- panel f []
```

Aknaraami *f* elemendi *p* paigutamiseks kasutatakse funktsiooni *set*.

```
    set f [ layout  := minsize (sz 300 300) $ widget p]
```

Sündmuste ja käitumiste loomiseks luuakse sündmuste võrgustik *networkDescription*, mille tüübiks on *MomentIO ()*.

```
let networkDescription :: MomentIO ()
```

Sündmuste võrgustikku luuakse sündmused *taimerE* ja *hiirE* [9]. *TaimerE* saab oma väärtuse funktsioonist *event0 :: w -> Event w (IO ()) -> MomentIO (Event ())*, kus esimeseks argumendiks on taimer *t* ning teiseks argumendiks *command*, mis on sündmus *WX* teegis. Tulemuseks on sündmus, mis toimub iga 10ms järel, kuna eelnevalt mainitud taimeri *t* intervall on 10ms.

HiirE saab oma väärtuse funktsioonist *event1 :: w -> Event w (a -> IO ()) -> MomentIO (Event a)*, kus esimeseks argumendiks on paneel *p* ning teiseks argumendiks *mouse* ehk tegemist on hiire sündmusega. Tulemuseks on hiire sündmus, mis toimub paneelil *p*.

```
networkDescription = do
    taimerE  <- event0 t command
    hiirE    <- event1 p mouse
```

Selleks, et kursori asukoha järgi muuta taustavärvi, loome funktsiooni *varv :: Point -> Color*. *Varv* on funktsioon, mis vastavalt kursori asukohale tagastab värvi kollane, roheline, punane või sinine.

Kursori *x* ja *y* asukoha määramiseks kasutatakse funktsioone *pointX :: Point -> Float* ja *pointY :: Point -> Float*, mis tagastavad vastavalt *x* ning *y* koordinaadi.

```
varv :: Point -> Color
varv x = if (pointX x > 150) && (pointY x > 150) then yellow
    else if (pointX x > 150) && (pointY x < 150) then green
    else if (pointX x < 150) && (pointY x > 150) then red
    else blue
```

Kuna hiire sündmustest on tähtis ainult liikumine, luuakse funktsioon *justMotion :: EventMouse -> Maybe Point*, mis tagastab *Just point* ehk kursori asukoha kursori liikumissündmuse puhul ning ülejäänud sündmuste puhul tagastatakse *Nothing* [9].

```
justMotion :: EventMouse -> Maybe Point
justMotion (MouseMotion pt _) = Just pt
justMotion _                  = Nothing
```

Taustavärvi muutmiseks luuakse käitumine *bcolor*, mille tüüp on *Behavior Color*. *Bcolor* saab oma väärtuse funktsioonist *stepper :: MonadMoment m => a -> Event a -> m (Behavior a)*, mille algväärtuseks on punane värv. Sündmuste vooks on *varv <$> (filterJust (justMotion <$> hiirE))*.

Igale sündmusele *hiirE* rakendatakse funktsiooni *fmap* abil varasemalt mainitud funktsiooni *justMotion*. Eraldamaks ainult *Just* tüüpi väärtusi, kasutatakse funktsiooni *filterJust :: Event (Maybe a) -> Event a*. *Just* tüüpi väärtustele rakendatatakse funktsiooni *fmap* abil funktsiooni *varv*, mis tagastab kursori asukohale vastava värvi.

```
(bcolor :: Behavior Color) <-
    stepper red (varv <$> (filterJust
                          (justMotion <$> hiirE)))
```

Kuna programmi eesmärgiks oli aknaraami *f* taustavärvi muutmine vastavalt kursori asukohale, siis funktsiooni *sink* abil määratakse aknaraami *f* taustavärviks ehk *bgcolor*'iks käitumine *bcolor*, mis vastavalt kursori positsioonile tagastab värvi.

```
sink f [bgcolor :== bcolor]
```

Aknaraamistiku uuendamiseks kasutatakse funktsioone *reactimate :: Event (IO ()) -> MomentIO ()* ja *repaint :: w -> IO ()* [9]. Funktsiooni *repaint* rakendatakse iga kord, kui toimub sündmus *taimerE*.

```
reactimate $ repaint f <$ taimerE
```

Kompileeritakse ning käivitatakse sündmustevõrgustik.

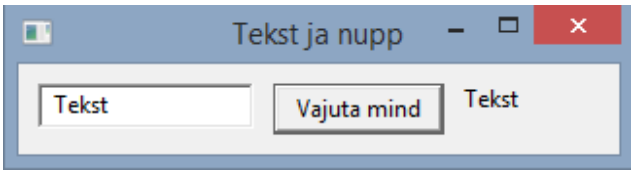
```
network <- compile networkDescription
actuate network
```

Viimasena tehakse meetod *main*, mis on *IO ()* tüüpi. *Main* meetodiga käivitatakse kasutajaliides.

```
main :: IO ()
main = start gui
```

3.5 Teksti väljastamine nupuvajutusel

Programmi eesmärgiks on lugeda kasutaja poolt sisestatud tekstiväljast väärtus ning nupuvajutuse korral väljastada see. Näidisprogrammi kood asub lisades (Lisa 4). Joonisel 11 on näidatud antud näidisprogrammi kasutajaliides.



Joonis 11. Näidisprogrammi 4 kasutajaliides

Esmalt luuakse kasutajaliides *gui*, mille tüübiks on *IO ()*. Kasutajaliidesesse lisatakse aknaraamistik *f*, kasutajale teksti sisestamiseks sisestusväli *input*, nupp *but* ja väljundi kuvamiseks staatiline tekstiväli *output*.

```
gui :: IO ()
gui = do
    f         <- frame [text := "Tekst ja nupp"]
    input     <- entry f []
    output    <- staticText f []
    but       <- button f [text := "Vajuta mind"]
```

Aknaraamistikku paigutatakse elemendid kasutades funktsiooni *set*. Elementideks on *input*, *but* ja *output*.

```
    set f [layout := margin 10 $ row 10 [widget input
        , widget but, minsize (sz 70 20) $ widget output]]
```

Järgmisena luuakse sündmuste võrgustik *networkDescription*, mille tüübiks on *MomentIO ()*.

```
    let networkDescription :: MomentIO ()
```

Sündmuste võrgustikku lisatakse käitumine *tekst* ja sündmus *but2*.

Käitumine *tekst* saab oma väärtuse funktsioonist *behaviorText*, mille esimene argument on sisestusväli *input* ning teine argument algväärtus ehk antud näite puhul tühi sõne.

Sündmus *but2* saab väärtuse funktsioonist *event0*, mille esimene argument on nupp *but* ning teine argument *command*. Tulemuseks on nupuvajutuse sündmus.

```
tekst <- behaviorText input ""
but2  <- event0 but command
```

Järgmisena luuakse käitumine nimega *behavior*, mille tüübiks on *Behavior String*.

Operaator *<@ :: Behavior b -> Event a -> Event b* võtab kaks argumenti, kus esimene argument on käitumine *Behavior b* ning teine argument sündmus *Event a*. Tulemuseks on sündmus *Event b*, mis on saadust käitumisest *Behavior b* sündmuse *Event a* toimumise ajahetkel.

Käitumine *behavior* saab oma väärtuse funktsiooni *stepper* abil, kus esimene argument ehk algväärtus on tühi sõne. Teine argument ehk sündmuste voog on *tekst <@ but2*, kus sündmused toimuvad *but2* sündmustega samaaegselt, kuid väärtus on saadud käitumise *tekst* väärtusest *but2* sündmuse toimumise ajahetkel. Käitumise *behavior* väärtuseks saab sisestusväljas olev väärtus nupu *but* vajutuse hetkel.

```
(behavior :: Behavior String)
    <- stepper "" (tekst <@ but2)
```

Programmi väljund kasutajaliidesega ühendatakse funktsiooni *sink* abil, mis võtab esimese argumendina staatilise tekstivälja *output* ning teises argumendis määratakse tekstiks käitumise *behavior* väärtus.

```
sink output [text :== behavior]
```

Järgmisena kompileeritakse võrgustik *networkDescription* ja käivitatakse see funktsiooniga *actuate*.

```
network <- compile networkDescription
actuate network
```

Viimasena luuakse funktsioon *main*, millega käivitatakse kasutaliides *gui*.

```
main :: IO ()
main = start gui
```

4. Kokkuvõte

Käesolevas lõputöös tutvustati funktsionaalset reaktiivset programmeerimist, loodi ning selgitati 4 näidisprogrammi.

Lõputöö teises peatükis tutvustati funktsionaalset reaktiivset programmeerimist, samuti selgitati 3 näidet sündmuste ja 3 näidet käitumiste kohta.

Kolmandas peatükis tutvustati *reactive-banana* teeki ja loodi neli näidisprogrammi, millega näidati *reactive-banana* teegi kõige tähtsamaid funktsioone ning mitmeid viise sündmuste ja käitumiste kasutamiseks ning kombineerimiseks.

Esimene näidisprogramm sisaldas kasutaja poolt sisestatud kahe arvu liitmist ning summa väljastamist. Teine näidisprogramm luges mitu korda on vajutatud nuppu ja väljastas nupuvajutuste summa ning paaris või paaritu oleku. Teine programm sisaldas nupuvajutuse sündmusi ja kahe käitumise kasutamist. Kolmas näidisprogramm, mis kasutas hiire ja taimeri sündmusi ning ühte käitumist, jälgis kursori positsiooni ning muutis vastavalt kursori asukohale kasutajaliidese taustavärvi. Neljas näidisprogramm, mis kasutas nupuvajutuse sündmusi ja kahte käitumist, luges kasutaja poolt sisestatud tekstiväljast väärtuse ning nupuvajutuse korral väljastas sisestatud väärtuse.

5. Kasutatud materjalid

[1] Elliott C, Hudak P. Functional reactive animation. In ACM SIGPLAN Notices 1997 Aug 1 (Vol. 32, No. 8, pp. 263-273). ACM.

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

[3] Hackage. The reactive-banana package. http://hackage.haskell.org/package/reactive-banana (10.05.2017)

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

[5] Staltz A. The introduction to Reactive Programming you've been missing. https://gist.github.com/staltz/868e7e9bc2a7b8c1f754 (09.05.2017)

[6] HaskellWiki. Functional Reactive Programming. https://wiki.haskell.org/Functional_Reactive_Programming (06.05.2017)

[7] Apfelmus H. Reactive-banana and the essence of FRP. http://apfelmus.nfshost.com/blog/2011/03/28-essence-frp.html (06.05.2017)

[8] HaskellWiki. FRP explanation using reactive-banana. https://wiki.haskell.org/FRP_explanation_using_reactive-banana (06.05.2017)

[9] Apfelmus H. Reactive-banana. https://github.com/HeinrichApfelmus/reactive-banana/tree/master/reactive-banana-wx/src (06.05.2017)

[10] Apfelmus H. Reactive-banana. https://wiki.haskell.org/Reactive-banana (06.05.2017)

6. Lisad

6.1 Lisa 1

Koodi saab kopeerida GitHubist - https://github.com/tennoveber/FRP_reactive_banana/blob/master/Arvude_liitmine.hs

```
import Data.Maybe
import Graphics.UI.WX hiding (Event)
import Reactive.Banana
import Reactive.Banana.WX

gui :: IO ()
gui = do
    f         <- frame [text := "Kahe arvu liitmine"]
    input1    <- entry f []
    input2    <- entry f []
    output    <- staticText f []

    set f [layout := margin 10 $ row 10
            [widget input1, label "+", widget input2
            , label "=", minsize (sz 40 20) $ widget output]]

    let networkDescription :: MomentIO ()
        networkDescription = do

           binput1  <- behaviorText input1 ""
           binput2  <- behaviorText input2 ""

           let
               readNumber :: String -> Maybe Int
               readNumber s = listToMaybe [x | (x,"") <- reads s]

               bmayint1,bmayint2 :: Behavior (Maybe Int)
               bmayint1 = readNumber <$> binput1
               bmayint2 = readNumber <$> binput2

               addNumber :: Maybe Int -> Maybe Int -> Maybe Int
               addNumber (Just a) (Just b) = Just (a+b)
               addNumber _ _ = Nothing

               bsum :: Behavior (Maybe Int)
               bsum = addNumber <$> bmayint1 <*> bmayint2

               showNumber :: Maybe Int -> String
               showNumber (Just a) = show a
```

```
            showNumber Nothing = "--"

            result :: Behavior String
            result = showNumber <$> bsum

        sink output [text :== result]
    network <- compile networkDescription
    actuate network
main :: IO ()
main = start gui
```

6.2 Lisa 2

Koodi saab kopeerida GitHubist - https://github.com/tennoveber/FRP_reactive_banana/blob/master/Nupuvajutus.hs

```
{-# LANGUAGE ScopedTypeVariables #-}
import Graphics.UI.WX hiding (Event)
import Reactive.Banana
import Reactive.Banana.WX
gui :: IO ()
gui = do
    f         <- frame [text := "Nupuvajutus"]
    nupp      <- button f [text := "+1"]
    valjund   <- staticText f []
    valjund2  <- staticText f []

    set f [layout := minsize (sz 350 40) $ margin 10 $ row 10 [widget nupp
        , label " Summa: ", widget valjund, widget valjund2]]
    let networkDescription :: MomentIO ()
        networkDescription = do

            nuppE   <- event0 nupp   command
            (behavior1 :: Behavior Int) <- accumB 0 ((+1) <$ nuppE)
            let
                behavior2 :: Behavior String
                behavior2 = func <$> behavior1
                    where
                    func x = if even x then "Tegemist on paarisarvuga"
                          else "Tegemist on paaritu arvuga"
            sink valjund [text :==  show <$> behavior1 ]
            sink valjund2 [text :==  show <$> behavior2 ]

    network <- compile networkDescription
    actuate network
```

```
main :: IO ()
main = start gui
```

6.3 Lisa 3

Koodi saab kopeerida GitHubist - https://github.com/tennoveber/FRP_reactive_banana/blob/master/Kursori_liigutamine.hs

```
{-# LANGUAGE ScopedTypeVariables #-}
import Graphics.UI.WX hiding (Event)
import Reactive.Banana
import Reactive.Banana.WX
gui :: IO ()
gui = do
    f  <- frame [ text       := "Hiire liikumine"
                , resizeable := False ]
    t  <- timer f [ interval := 10 ]
    p  <- panel f []
    set f [ layout  := minsize (sz 300 300) $ widget p]

    let networkDescription :: MomentIO ()
        networkDescription = do
            taimerE  <- event0 t command
            hiirE    <- event1 p mouse

            (bcolor :: Behavior Color) <-
                stepper red (varv <$> (filterJust (justMotion <$> hiirE)))

            sink f [bgcolor :== bcolor]
            reactimate $ repaint f <$ taimerE
    network <- compile networkDescription
    actuate network

main :: IO ()
main = start gui

varv :: Point -> Color
varv x = if (pointX x > 150) && (pointY x > 150) then yellow
    else if (pointX x > 150) && (pointY x < 150) then green
    else if (pointX x < 150) && (pointY x > 150) then red
    else blue

justMotion :: EventMouse -> Maybe Point
justMotion (MouseMotion pt _) = Just pt
justMotion _                  = Nothing
```

6.4 Lisa 4

Koodi saab kopeerida GitHubist - https://github.com/tennoveber/FRP_reactive_banana/blob/master/Tekst_ja_nupp.hs

```
{-# LANGUAGE ScopedTypeVariables #-}
import Graphics.UI.WX hiding (Event)
import Reactive.Banana
import Reactive.Banana.WX

gui :: IO ()
gui = do
    f         <- frame [text := "Tekst ja nupp"]
    input     <- entry f []
    output    <- staticText f []
    but       <- button f [text := "Vajuta mind"]

    set f [layout := margin 10 $ row 10 [widget input
        , widget but, minsize (sz 70 20) $ widget output]]

    let networkDescription :: MomentIO ()
        networkDescription = do

            tekst <- behaviorText input ""
            but2  <- event0 but command

            (behavior :: Behavior String)
                <- stepper "" (tekst <@ but2)

            sink output [text :== behavior]

    network <- compile networkDescription
    actuate network

main :: IO ()
main = start gui
```

6.5 Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Tenno Veber**,
(*autori nimi*)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   Haskelli FRP teegi uurimine: Reactive-banana,
   (*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**