

VILNIAUS UNIVERSITETAS  
MATEMATIKOS IR INFORMATIKOS FAKULTETAS  
INFORMATIKOS KATEDRA

## **Funkcinio-reaktyvaus programavimo taikymas įvykių kaupimo sistemose**

### **Functional Reactive Programming in Event Sourcing Systems**

Mokslo tiriamasis darbas

Atliko: 1 kurso 12 grupės studentas

Žilvinas Kučinskas .....  
(parašas)

Darbo vadovas:

Viačeslav Pozdniakov .....  
(parašas)

Recenzentas:

prof. Rimantas Vaicekuskas .....  
(parašas)

Vilnius, 2014 m.

## Turinys

1. Magistro darbo objekto apžvalga bei tyrimo problemos aprašymas .....	3
1.1. Tyrimo objektas .....	3
1.2. Darbo tikslai ir uždaviniai .....	3
1.3. Tyrimo aktualumas .....	3
1.4. Pritaikymo pavyzdys .....	5
1.5. Tyrimo metodika .....	6
1.6. Laukiami rezultatai .....	7
2. Literatūros analizė .....	8
2.1. Funkcinis-reaktyvus programavimas .....	8
2.1.1. Įvadas .....	8
2.1.2. Pagrindinis tiklas .....	8
2.1.3. Sąvokos .....	8
2.1.4. Sąvybės .....	9
2.1.5. Įvykių srautas .....	9
2.1.6. Įvykių srautas funkciniam programavime .....	10
2.2. Įvykių kaupimas .....	15
2.2.1. Įvadas .....	15
2.2.2. Momentinė kopija .....	17
2.2.3. Įvykių kaupimo privalumai ir trūkumai .....	17
2.2.4. Įvykių kaupimas funkciniam programavime .....	18
2.3. Monados .....	18
2.3.1. Terminas .....	18
2.3.2. Taisyklės .....	19
2.3.3. Tolesnis darbas .....	19
2.4. Išvados .....	20
3. NAUJA!!!! .....	21
3.0.1. Random intro .....	21
3.0.2. Klasikinis FRP .....	22
3.0.3. Klasikinio FRP istorija .....	22
3.0.4. Dabartinės klasikinės FRP sistemos .....	23
3.0.5. Signalo funkcijos FRP .....	24
3.0.6. Neįvykdyti iššūkiai .....	25
3.0.7. Naudota čia .....	26
3.0.8. Fran Abstract .....	26
3.0.9. Įvadas Fran .....	27
3.0.10. Formali semantika .....	30
3.0.11. Semantinės domeno sritys .....	30
3.0.12. Semantinės funkcijos .....	31
3.0.13. Elgesio semantika .....	32
4. Sąvokų apibrėžimai .....	33
5. Santrumpos ir paaiškinimai .....	34
Literatūros sąrašas .....	35

# **1. Magistro darbo objekto apžvalga bei tyrimo problemos aprašymas**

## **1.1. Tyrimo objektas**

Tyrimo objektas yra funkcinio-reaktyvaus programavimo bei įvykių kaupimo principai.

## **1.2. Darbo tikslai ir uždaviniai**

Darbo tikslas yra išnagrinėti funkcinio-reaktyvaus programavimo pritaikymo galimybes įvykių kaupimo sistemose.

Siekiant šio tikslo, turi būti išspręsti šie uždaviniai:

- įrodyti, kad funkcinį-reaktyvų programavimą įmanoma taikyti įvykių kaupimo sistemose;
- sukurti konkretizuotą kalbą (angl. domain specific language), apjungiančią funkcinio-reaktyvaus programavimo bei įvykių kaupimo principus;
- aprašyti konkretizuotuos kalbos kūrimo metodiką, apibrėžti gautų rezultatų apribojimus, suformuluoti iškilusias problemas bei paaiškinti jų priežastis.

## **1.3. Tyrimo aktualumas**

Funkcinis reaktyvus programavimas integruoja laiko tėkmę bei sudėtinius įvykius į funkcinį programavimą. Šis principas suteikia elegantišką būdą išreikšti skaičiavimus interaktyvių animacijų, robotikos, kompiuterinio vaizdavimo, vartotojo sąsajos ir modeliavimo srityse [Cza12, p. 4]. Pagrindinės funkcinio reaktyvaus programavimo sąvokos:

- signalai arba elgsena - reikšmės, besikeičiančios bėgant laikui;
- įvykiai - momentinių reikšmių kolekcijos arba laiko-reikšmės poros.

Funkcinis-reaktyvus programavimas įgalina apsirašyti elgseną deklaratyviai naudojant imperatyvios programavimo kalbos struktūras [EH97, p.1]. Elgsena ir įvykiai

gali būti komponuojami kartu, išreikšti vienas per kitą. Funkcinis reaktyvus programavimas apibrėžia kaip signalai arba elgsena reaguoja į įvykius. [Ams, p. 1] Šį principą galima iliustruoti pavyzdžiu. Tarkime turime Excel lapą, kuriame yra trys laukai: darbuotojo pradirbtos valandos, valandinis užmokestis bei formulė, kuri paskaičiuoja konkretų darbuotojo atlyginimą. Darbuotojui pradirbus daugiau valandų, atnaujinamas pradirbtų valandų skaičius. Kartu atsinaujina ir pačios formulės reikšmė, tai yra konkretus užmokestis pakinta. Šiuo atveju įvykus reikšmės atnaujinimo įvykiui, nuo jos priklausomos formulės taipogi atsinaujina arba tam tikras įvykis iššaukia elgseną sistemoje.

Įvykių kaupimo principo esmė – objektas yra atvaizduojamas kaip įvykių seka. Kaip pavyzdį tai galima parodyti remiantis banko sąskaita. Tarkime vartotojas, banko klientas, turi 100 litų sąskaitos balansą. Tarkime vartotojas nusipirko prekę už 20 litų, tada įnešė į savo sąskaitą 15 litų ir galiausiai nusipirko tam tikrą paslaugą už 30 litų. Akivaizdu, jog turint šią įvykių seką, galima atvaizduoti dabartinę objekto būseną - tai yra 65 litai vartotojo sąskaitoje. Įvykių kaupimo principas užtikrina, jog visi būsenos pasikeitimai yra saugomi įvykių žurnale kaip įvykių seka [Ver13]. Įvykių kaupimo principui yra būdinga, jog įvykių negalima ištrinti bei atnaujinti, duomenys yra nekeičiami, dėl to įvykių žurnalas yra sistemos gyvavimo istorija (tiesos šaltinis). Tačiau toks modelis turi ir trūkumų. Jis nėra pritaikytas patogiam užklausų rašymui. Iš įvykių srautų yra kuriamos projekcijos, skirtos konkretiems sistemoms poreikiams, pavyzdžiui: paieškai, klasifikacijai ar ataskaitų ruošimui.

Pritaikius funkcinį-reaktyvų programavimą įvykių kaupimo principu paremtose sistemose būtų galima modeliuoti ne tik momentinius įvykius, tačiau turėti ir jų istoriją. Yra poreikis sukurti konkretizuotą kalbą (angl. domain specific language), kuri įgalintų paslėpti įvykių žurnalą (arba duomenų saugyklą). Pastarosios naudotojas galėtų orientuotis į pačią sprendžiamos srities problemą, nekreipdamas dėmesio į žemesnio lygio realizacijos detales. Šiuo atveju viename faile būtų galima deklaratyviai (ką kažkuri programos dalis turi daryti) apsirašyti elgseną, nutikus įvykiui, kartu su imperatyviomis (instrukcijos, kurios aprašo, kaip programos dalys atlieka savo užduotis) struktūromis.

## 1.4. Pritaikymo pavyzdys

Tarkime turime domeno sritį - bankininkystė. Turime įvykių srautą - vartotojų sukūrimas. Naudojant įsivaizduojamą Scala API galima sukurti vartotojų paieškos puslapį pagal vardą ir pavardę. Demonstracija pateikta 3 kodo pavyzdyje.

```
// stream model
case class CustomerCreate (val name: String, val surname: String, val personalNum:
    String)

val es = EventSourceConnection(url)
val createStream = Stream(es, "customerCreate")

class case CustomerModel(val name: String, val surname: String, val personalNum:
    String) extends ViewableModel

trait CustomerArgModel extends Arg2Model[String,String]{
    val name: Option[String]
    val surname: Option[String]
}

//args are passed on form view/post
val customerView = View(args: CustomerArgModel).foldLeft(
    (acc,event) => event match {
        case CustomerCreate(name, surname, personalNum) =>
            for {
                newName <- name
                newSurname <- surname
                newPersNum <- personalNum
                if (args.name == name && args.surname == surname)
            } yield CustomerModel(newName, newSur, newPersNum)
    }
)

// getting data for all Kucinskai
val specificData = customerView(None, Some("Kucinskas")):
    Option[List[CustomerModel]]
```

---

Kodo pavyzdys 1 - vartotojų paieškos puslapis naudojant įsivaizduojamą Scala API

Šiuo atveju veiksmai peržiūrint duomenis yra sumaišomi kartu su veiksmiais gaunant įvykius. Verta pastebėti, jog lokali duomenų saugykla nebuvo paminėta arba apibrėžta. Pastaroji gali būti sugeneruota bei valdoma automatiškai.

Antruoju pavyzdiniu atveju turimas vartotojo balanso(įplaukiančios/išplaukiančios lėšos) įvykių srautas ir norima gauti vartotojo, kurio asmens kodas yra 39008226547, einamosios savaitės sąskaitos balansą. Demonstracija pateikta 2 kodo pavyzdyje.

```
val duration = 1.weeks
val personalNum = "39008226547"
val balanceStream = Stream(es, "customerBalance")
val notOlderThanOneWeek =
  for {
    event <- balanceStream
    filtered <- event.filter(_.personalNum == personalNum
      && (DateTime.now - _.timeStamp) >= duration)
  } yield filtered
val sum = notOlderThanOneWeek.toList.sum
```

Kodo pavyzdys 2 - vartotojo einamosios sąskaitos balansas naudojant įsivaizduojamą Scala API

Šiuo atveju *event* kintamasis yra įvykių srauto monada (terminas vartojamas funkciniam programavimui, kilęs iš kategorijų teorijos ir turi savas taisykles), o *filtered* kintamasis - duomenų saugyklos monada. Bendruoju atveju skirtingos monados tarpusavyje nesiderina [KW92]. Dėl to reikia išsiaiškinti - ar yra įmanoma ir kaip šias monadas suderinti.

## 1.5. Tyrimo metodika

Darbo analitinėje dalyje bus naudojami tradiciniai bibliotekinio tyrimo metodai. Darbo tikslui pasiekti tiriamojoje dalyje bus pasirinkta konkreti funkcinė programavimo kalba (pvz.: Haskell, Scala) bei aprašoma kūrimo metodika.

Papildomai magistro darbo rašymo metu ketinama užbaigti Erik Meijer, Martin Odersky, Roland Kuhn dėstomą reaktyvaus programavimo kursą (Coursera). Esant galimybėms žadama apsilankyti į funkcinį programavimą orientuotose konferencijose Scala Days (<http://scaladays.org/>), Scalar (<http://scalar-conf.com/>) ar kitose.

## **1.6. Laukiami rezultatai**

Magistrinio darbo metu planuojama išnagrinėti funkcinio-reaktyvaus programavimo ir įvykio kaupimo principus, įrodyti, jog šie principai gali būti panaudoti kartu bei suderinti, sukurti konkretizuotą kalbą (angl. domain specific language), apjungiančią šiuos principus, bei aprašyti kūrimo eigos metodiką, apibrėžti gautus rezultatus, suformuluoti apribojimus, iškilusias problemas bei paaiškinti jų priežastis.

## 2. Literatūros analizė

### 2.1. Funkcinis-reaktyvus programavimas

#### 2.1.1. Įvadas

Anot [Ams], funkcinis-reaktyvus programavimas, toliau dažnai vadinamas tiesiog FRP, yra būdas modeliuoti reaktyvų - besikeičiančius laiko tekmeje bei reaguojančius į išorinį stimulą - elgesį visiškai funkcinėse programavimo kalbose. FRP leidžia deklaratyviu ir paprastu būdu modeliuoti sistemas, kurios turi reaguoti į duomenis bėgant laikui.

#### 2.1.2. Pagrindinis tiklas

Pagrindinis funkcinio-reaktyvaus programavimo tikslas [Ams]:

- saugus programavimas - kompiliatorius turi kiek įmanoma patikrinti programų korektiškumą;
- efektyvus programavimas - programos turėtų veikti realiu laiku, todėl efektyvios ir optimizuotos operacijos yra būtinos;
- komponavimas - FRP leidžia kurti programas iš smulkesnių programų, o ne orientuotą į problemą, vientisą kodą.

#### 2.1.3. Sąvokos

Pagrindinės FRP sąvokos yra:

- signalai arba elgsena - besikeičiančios laike reikšmės;
- įvykiai - kolekcija momentinių reikšmių arba laiko-reikšmės poros.

FRP pasiekia reaktyvumą naudodamas konstrukcijas, kurios tiksliai apibrėžia kaip signalai arba elgsena pasikeičia reaguodami į įvykius. Tai yra pagrindinis būdas išreiškiant bei realizuojant elgseną. Kitu būdu, elgsena gali būti laiko semantinės funkcijos<sup>1</sup>, kuriose laikas yra pakeičiamas kilus įvykiui [NCP02].

---

<sup>1</sup><http://msdl.cs.mcgill.ca/people/tfeng/docs/as/node5.html>



#### 2.1.4. Sąvybės

Anot anksčiausios funkcinio-reaktyvaus programavimo formuluotės [EH97], pagrindinės sąvybės, kuriomis pasižymi FRP:

- elgsenos arba signalų modeliavimas bėgant laikui,
- įvykių, kurie turi baigtinį skaičių atsitikimų daugelyje laiko taškų, modeliavimas,
- perjungimas (angl. switching) - sistema gali pasikeisti dėl atsitikusių įvykių,
- analizės detalių, tokių kaip reaktyvaus modelio įvykių ėmimo dažnis, atskyrimas.

#### 2.1.5. Įvykių srautas

Pagal [Bas], įvykių srautas yra eilė pagal laiką surikiuotų įvykių, pavyzdžiui akcijų rinkos srautas.

Įvykių srautas kaip duomenų srauto tipas formaliai atrodo kaip pora  $(s, t)$ , kur  $s$  yra seka surikiuotų sąrašo įvykių, o  $t$  yra seka laiko intervalų ir kiekvienas intervalas yra netuščias.

Tokio duomenų srauto pavyzdžiai gali būti:

- akcijų kursas,
- paspaudimų srautas,
- tinklo srautas,
- GPS<sup>2</sup> duomenys.

Įvykių srauto apdorojimas pagal atsitikimo laiką turi privalumų:

- įvykių apdorojimo algoritmai naudoja mažai atminties, nes jiems nereikia prisiminti daug įvykių;
- algoritmai gali būti labai greiti;

---

<sup>2</sup>[http://en.wikipedia.org/wiki/Global\\_Positioning\\_System](http://en.wikipedia.org/wiki/Global_Positioning_System)

- gavus įvykį, skaičiavimai atliekami iškart, todėl galima perduoti rezultatą kitam skaičiavimui ir pamiršti įvykį.

Įvykių srauto apdorojimas labiau akcentuoja didelio našumo duomenų gavimą ir matematinių algoritmų pritaikymą įvykių duomenims. Taip pat įvykių srautai įprastai pritaikomi konkrečiai sistemai ar organizacijai.

### 2.1.6. Įvykių srautas funkciname programavime

Vietoje įvykių srauto, galima naudoti klausytojo projektavimo šabloną [OSG04], tačiau ilgame programinės įrangos kūrimo gyvavimo cikle įvykių srauto naudojimas palengvina kai kuriuos dalykus. Pavyzdžiui, imperatyvaus programavimo atžvilgiu, viename metode turime klausytoją, kuris reaguoja į situaciją *A*, tačiau iškilus situacijai *B*, šis klausytojas yra pašalinamas. Šiuo atveju programinis kodas, kuris valdo klausytojo gyvavimo ciklą yra įsipynęs keliose skirtingose kodo vietose, ko pasekoje tai reiškia, jog yra sunkiau palaikyti, stebėti, keisti bei suprasti šias vietas. Įvykių srautas yra pirmosios klasės reikšmės, todėl jį galima abstrahuoti [MO12]. Abstrahuojant klausytojų gyvavimo ciklo valdymo idėją tampa lengviau programuoti pagal tai kas turėtų nutikti, o ne pagal tai, ką kompiuteris turėtų daryti toliau. Tačiau turbūt didesnis įvykių srauto pranašumas yra tai, jog pastarasis gali būti transformuojamas ir komponuojamas. Toliau bus aprašomi operacijos arba veiksmi bei sutrumpinti išeities kodo pavyzdžiai, kuriuos galima atlikti su įvykių srautu remiantis Reactive Web<sup>3</sup> karkasu, skirtu Scala<sup>4</sup> programavimo kalbai.

Pirmas veiksmas - įvykio srauto sukūrimas. Jis pavaizduotas 3 kodo pavyzdyje. Pirmiausia sukuriamas įvykių šaltinis, kuris po to priskiriamas įvykių srautui.

```
val eventSource = new EventSource[String]{}
scheduleTask(10000) {
    eventSource.fire("Event after 10 seconds")
}

val eventStream: EventStream[String] = eventSource
```

Kodo pavyzdys 3 - įvykių srauto sukūrimas

<sup>3</sup><http://scalareactive.org>

<sup>4</sup><http://www.scala-lang.org>

Taip pat *EventSource* (įvykių šaltinis) turi naudingą poklasį *Timer* (laikrodis). Jo realizacija pavaizduota 4 kodo pavyzdyje. Šis sukuria tikslėjimo įvykius duotu laiko intervalu.

```
val timer = new Timer(0, 2000, {t => t >= 32000})

for(t <- timer)
yield "timer: " + t.toString
```

#### Kodo pavyzdys 4 - įvykių srauto sukūrimas

Įvykių srautas valdo kolekciją klausytojų, tačiau konceptualiai reikėtų mąstyti kitaip. Klausytojo pridėjimas iš tikrųjų reiškia funkcijos iškvietimą kiekvienam kilusiui įvykiui, kitais žodžiais kiekvienam įvykių srauto įvykiui. Lygiai taip pat kaip Scala programavimo kalboje norint įvykdyti funkciją kiekvienai kolekcijos reikšmei yra iškviečiama *foreach* funkcija. Tačiau įvykių srauto atveju, *foreach* grąžina rezultatą akimirksniu, o funkcija yra išsaugoma ir vėliau įvykdoma kai tik iššaukiamas koks nors įvykis. Šio atvejo praktinis variantas pademonstruotas 5 kodo pavyzdyje.

```
val eventSource = new EventSource[String] {}

// The following is syntactic sugar for
// eventSource.foreach(event => alert("You fired: " + event + ""))
for(event <- eventSource) {
    alert("You fired: " + event + "")
}
```

#### Kodo pavyzdys 5 - klausytojų pridėjimas

Rinkinys transformacijos metodų įvykių srautą padaro labai universaliu. Šios transformacijos grąžina modifikuotą įvykių srautą. Transformacijas galima atlikti viena po kitos. Tai primena transformacijas galimas su Scala kolekcijomis, pavyzdžiui 6 kodo pavyzdyje. Kai tik originalus įvykių srautas gauna įvykį, transformuotas įvykių srautas iššauka savo įvykį pritaikant klausytojo funkciją.

```
List(1,2,3).map(__ * 10).filter(__ < 25)
```

#### Kodo pavyzdys 6 - kolekcijos transformacijos

Jeigu įvykių srautas gauna labai daug įvykių, tačiau aktualūs yra tik dalis jų, galima naudoti filtravimą, tai yra *filter* metodą. Tai demonstruojama 7 kodo pavyzdyje. Čia kiekvienam įvykiui yra pritaikomas predikatas. Jeigu jis įvertinimas kaip teisingas arba kitaip *true*, įvykis yra iššaukiamas transformuotame įvykių sraute.

```
val eventSource = new EventSource[String] {}  
eventSource.filter (_.length < 5)
```

#### Kodo pavyzdys 7 - įvykių srauto filtravimas

Kitas esminis kolekcijų metodas yra *map*. Jis leidžia transformuoti kolekciją pritaikant tam tikrą funkciją kiekvienam kolekcijos elementui. Pritaikius yra grąžinama nauja modifikuota kolekcija. Lygiai taip pat šis metodas gali būti pritaikytas įvykių srautui. Šis atvejis demonstruojamas 8 kodo pavyzdyje.

```
val eventSource = new EventSource[String] {}  
eventSource.map(_.reverse)
```

#### Kodo pavyzdys 8 - įvykių visiškas transformavimas

Įvykių srautui taip pat galima pritaikyti *flatMap* metodą. Scala kolekcijose šis metodas pritaiko funkciją visiems elementams, kuri grąžina naują kolekciją, kurioje sujungiamos visos grąžintų kolekcijų reikšmės. Pavyzdys pateiktas kodo pavyzdyje 9. Operacijos *map* atveju būtų grąžinta reikšmė *List(List(10, 11, 12), List(20, 21, 22), List(30, 31, 32))*, tačiau operacija *flatMap* sujungia grąžintų kolekcijų reikšmes.

```
val original = List(1, 2, 3)  
val flatMapped = original.flatMap(x => List(x*10,x*10+1,x*10_2))  
flatMapped == List(10,11,12, 20,21,22, 30,31,32)
```

#### Kodo pavyzdys 9 - flatmap Scala kolekcijose

Panašiai veikia *flatMap* metodas įvykių srautuose. Pastarasis leidžia sukurti įvykių srautą, kuris iššaukia įvykius, kurie yra iššauti kitų įvykių srautų. Metodas demonstruojamas 10 kodo pavyzdyje. Čia figūra yra išplečiama, o po to išnyksta.

```
// Assuming Shape is a case class with scale and opacity values  
// and millisTimer fires events once per millisecond, starting  
// at zero.  
// Scale should animate from 0 to 1 over the first second,
```

```
// and opacity should animate from 1 to 0 over the next.
def compositeAnimation(millisTimer: EventStream[Long], shape: Shape):
  EventStream[Shape] = {
    val scale: EventStream[Double] =
      millisTimer.map(m => m/1000.0)
    val opacity: EventStream[Double] =
      millisTimer.map(m => 1 - (m-1000)/1000.0)
    val seconds = millisTimer.filter(_ % 1000 == 0).map(_ / 1000).
      takeWhile(_ < 2000)

    seconds.flatMap {
      case 0 => scale
      case 1 => opacity
    }
  }
}
```

#### Kodo pavyzdys 10 - flatmap įvykių sraute

Imperatyviose kalbose, dažna užduotis yra iteruoti per masyvą ir atlikti tam tikrus veiksmus. Šiam veiksmui atlikti naudojamas tam tikras skaičius kintamųjų veiksmam atlikti, pavyzdžiui skaičiavimam. Funkciniame programavime šiam tikslui dažnai yra naudojamas *foldLeft* metodas. Pastarasis priima pradinę reikšmę arba kitaip būseną ir funkciją, kuri priima paskutinę reikšmę arba būseną ir kitą kolekcijos elementą. Kiekvienam elementui funkcija grąžina reikšmę arba būseną, kuri naudojama sekančiam funkcijos kvietimui. Verta pastebėti, jog funkcinis programavimas čia leidžia kodą vykdyti lygiagrečiai jo visiškai nekeičiant. 11 kodo pavyzdys demonstruoja sąrašo elementų sumos skaičiavimą.

```
list.foldLeft(0){(totalSoFar, nextElement) => totalSoFar + nextElement}
//more commonly written as list.foldLeft(0)(_ + _)
```

#### Kodo pavyzdys 11 - sąrašo elementų skaičiavimas naudojant flatmap

Panašiai *foldleft* metodą galima pritaikyti įvykių srautam. Demonstracija 12 kodo pavyzdyje.

```
case class AvgState(total: Double, count: Int)
```

```

val eventSource = new EventSource[String] {}

eventSource.foldLeft(AvgState(0,0)){
  case (AvgState(total, count), s) => AvgState(total+s.length,count+1)
} map {
  case AvgState(total, count) =>
    "Average length so far: " + (total/count)
}

```

#### Kodo pavyzdys 12 - foldleft metodas įvykių sraute

Įvykių srautus galima ne tik transformuoti, bet ir komponuoti, tai yra sujungti. Demonstracija 13 kodo pavyzdyje. Įvykių srautas *allClicks* išsauna įvykį kai bet kuris iš kitų įvykių srautų išsauna įvykį.

```

val allClicks = leftClicks | middleClicks | rightClicks

```

#### Kodo pavyzdys 13 - įvykių srautų sujungimas

Įvykių srautas atvaizduoja įvykius kaip srautą diskrečių reikšmių laiko tekmeje, taigi šios reikšmės egzistuoja momentiškaiai. Tačiau įvykių srautą taip pat galima paversti signalu. Signalas atvaizduoja besitęsiančią reikšmę, tai yra turi dabartinę reikšmę. Kai signalo reikšmė pasikeičia, įvykių srautas išsauna naują įvykį. Įvykių srautą norint paversti signalu galima naudoti *hold* metodą, kuriam yra paduodama pradinė reikšmė. Demonstracija 14 kodo pavyzdyje.

```

val eventSource = new EventSource[String] {}
val signal = eventSource.hold("(initial value of signal)")

def render = Demos.eventSourceInput(eventSource) ++
  <p>Signal value: '{Demos.signalOutput(signal)}'</p>

```

#### Kodo pavyzdys 14 - įvykių srautų pavertimas signalu

## 2.2. Įvykių kaupimas

Šiame skyriuje yra aprašomos žinios apie įvykių kaupimą, plusus ir minusus, įvykių srautus bei įvykių kaupimą funkciniam programavime remiantis daugiausia Vaughn Vernon surinkta ir aprašyta informacija [Ver13].

### 2.2.1. Įvadas

Kartais verslui svarbu fiksuoti objekto pasikeitimus domeno modelyje<sup>5</sup>. Šiuos pasikeitimus galima stebėti skirtingais būdais. Įprastai yra pasirenkama stebėti kai esybė<sup>6</sup> yra:

- sukurta,
- paskutinį kartą modifikuota
- bei kas atliko modifikaciją.

Tačiau šis būdas nepateikia jokios informacijos apie vienkartinį pasikeitimą.

Atsiradus poreikiui stebėti pasikeitimus detaliau, verslas reikalauja dar daugiau metaduomenų<sup>7</sup>, ko pasekoje tokie faktai kaip individualios operacijos laiko tecomėje bei jų įvykdymo laikas tampa svarbūs. Šie poreikiai verčia įvesti audito žurnalą fiksuoti labai tikslias panaudojimo atvejų metrikas, tačiau pastarasis būdas turi apribojimų. Jis gali atskleisti dalį informacijos apie tai kas nutiko sistemoje, leisti rasti bei ištaisyti dalį riktų bei klaidų programinėje įrangoje. Bet audito žurnalas neleidžia patikrinti domeno objekto būsenos prieš ir po tam tikrų pasikeitimų. O jeigu būtų galima išgauti daugiau informacijos iš pasikeitimų stebėjimo?

Visi programinės įrangos kūrėjai susiduria su labai tikslu pasikeitimų stebėjimu. Įprastas ir populiarus pavyzdys yra išeities kodo saugyklos, tokios kaip CVS<sup>8</sup>, Subversion<sup>9</sup>, Git<sup>10</sup> arba Mercurial<sup>11</sup>. Visos šios pataisų valdymo sistemos leidžia stebėti pirminių failų pasikeitimus. Įrankiai leidžia peržiūrėti išeities kodo artefaktus nuo

---

<sup>5</sup>[http://en.wikipedia.org/wiki/Domain\\_model](http://en.wikipedia.org/wiki/Domain_model)

<sup>6</sup><http://en.wikipedia.org/wiki/Entity>

<sup>7</sup><http://en.wikipedia.org/wiki/Metadata>

<sup>8</sup><http://www.nongnu.org/cvs/>

<sup>9</sup><http://subversion.apache.org/>

<sup>10</sup><http://git-scm.com/>

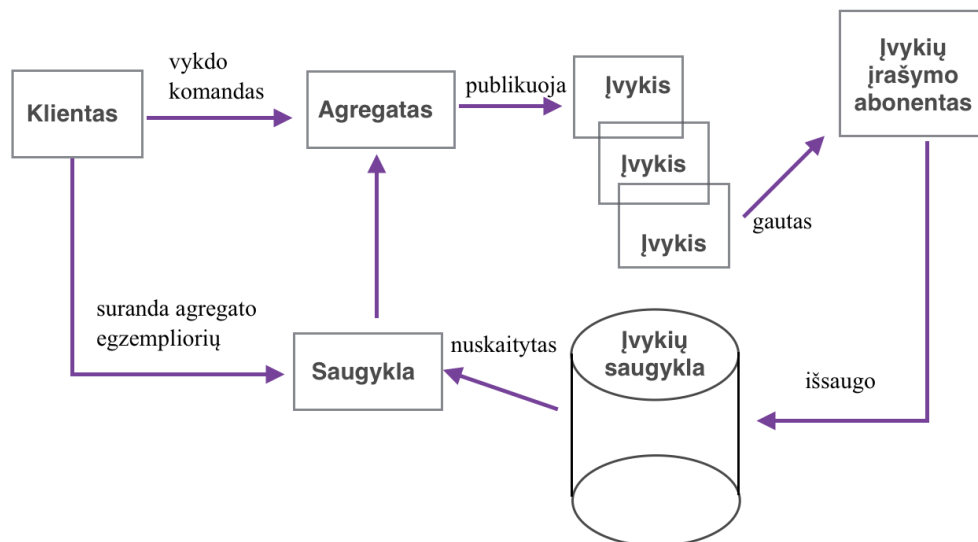
<sup>11</sup><http://mercurial.selenic.com/>

pačios pirmosios pataisos iki paskutinės. Kai visi išeities failai yra nusiunčiami į pataisų kontrolės sistemą, ši gali stebėti pasikeitimus viso programinės įrangos kūrimo gyvavimo ciklo metu.

Jeigu šis principas būtų pritaikytas vienai esybei, tada vienam agregatui<sup>12</sup> bei galiausiai kiekvienam modelio agregatui, galima suprasti kokią naudą atneša sistemos objektų pasikeitimų stebėjimas:

- Kas būtų nutiko modelyje, jog agregato egzempliorius buvo sukurtas?
- Kas nutiko agregato egzemplioriui bėgant laikui? (Operacijų požiūriu)

Turint visų atliktų operacijų istoriją, galima palaikyti laikinus modelius. Toks kaitos stebėjimas yra įvykių kaupimo principas. 1 diagramoje pateikia šio principo aukšto lygio reprezentaciją. Agregatai publikuoja įvykius, kurie yra išsaugomi įvykių saugykloje ir naudojami sekti modelio būsenos pasikeitimus. Verta paminėti, jog įvykiai reprezentuoja tam tikrą būsenos pasikeitimą bėgant laikui, todėl jie nėra atnaujinami arba ištrinami. Saugykla nuskaito įvykius iš įvykių saugyklos ir pritaiko juos vieną po kito taip atkurdamą agregato būseną.



1 pav. Įvykių kaupimo aukšto lygio reprezentacija

<sup>12</sup>[http://martinfowler.com/bliki/DDD\\_Aggregate.html](http://martinfowler.com/bliki/DDD_Aggregate.html)



### 2.2.2. Momentinė kopija

Ilgame periode sistemoje susikaupia daugybė įvykių. Atkuriant agregato būseną reikia atkartoti šimtus, tūkstančius ar net milijonus įvykių. Tai tampa šio modelio silpnąja puse, nes įvykių atkartojimas užtrunka vis ilgiau sistemai plečiantis.

Tačiau šio duomenų kamsčio galima išvengti naudojant agregato būsenos momentines kopijas. Tam tikrame įvykių saugyklos istorijos taške yra padaroma agregato būsenos kopija. Serializuota agregato būseną yra įrašoma į įvykių saugyklą. Nuo to momento, agregatas yra atkuriamas pirmiausia naudojantis naujausia jo būsenos momentine kopija ir tik po to atkartojami visi naujesni įvykiai.

Momentinės kopijos nėra atkuriamos atsitiktinai. Jos gali būti kuriamos kas apibrėžtą skaičių įvykių. Šis skaičius turėtų būti parinktas analizuojant domeno sritį bei sistemą ir radus optimalų variantą. Tikėtina, gali būti 50 arba 100 įvykių tarp momentinių kopijų.

### 2.2.3. Įvykių kaupimo privalumai ir trūkumai

Kaip saugojimo mechanizmas, įvykių kaupimas stipriai skiriasi ir pakeičia ORM<sup>13</sup> įrankį. Kadangi įvykiai dažnai įrašomi kaip dvejetainės reprezentacijos, jie negali būti optimaliai naudojami užklausoms atlikti. Faktiškai įvykių kaupimu pagrįstoms saugykloms tereikia vienos operacijos - gauti įrašus pagal unikalią agregato tapatybę [You10]. To pasekoje užklausa daryti reikia kito kelio. Dažniausiai tam pasirenkamas CQRS<sup>14</sup> principas [BDM<sup>+</sup>13].

Įvykių kaupimas verčia kitaip mąstyti apie domeno modelį. Įvykių istorija gali padėti surasti bei ištaisyti sistemos defektus bei klaidas [Fit12]. Derinimas naudojant istoriją visų veiksmų, kurie nutiko sistemoje, turi didžiulį pranašumą. Įvykių kaupimas gali vesti prie didelio našumo domeno modelių, tai yra palaikyti ypač didelį skaičių operacijų per sekundę. Pavyzdžiui, įrašymas į vieną duomenų saugyklos lentelę yra ypač greitas. Negana to, tai leidžia CQRS užklausų modelį išplėsti horizontaliai, nes duomenų šaltinio atnaujinimai įvykdomi fone, kai įvykių saugykla yra atnaujinama naujais įvykiais.

---

<sup>13</sup><http://www.orm.net/>

<sup>14</sup><http://martinfowler.com/bliki/CQRS.html>

#### 2.2.4. Įvykių kaupimas funkciniam programavime

Vaughn Vernon pateikia keletą pastebėjimų apie įvykių kaupimą funkciniam programavime, kurie gali būti naudingi atliekant projektinius sprendimus bei eksperimentinį tyrimą:

- Agregatas projektuojamas kaip nekintantis būsenos įrašas kartu su funkcijomis, kurios keičia būseną. Šios funkcijos paprasčiausiai priima būsenos įrašą ir įvykių argumentus ir gražina naują būsenos įrašą kaip rezultatą. Tokia funkcija pavaizduota 15 kodo pavyzdyje.

Funkcija<Busena, Ivykis, Busena>

Kodo pavyzdys 15 - agregato būsenos keitimas

- Dabartinė agregato būseną gali būti apibrėžta kaip suskleidimas į kairę visų praeities įvykių, kurie yra perduodami būseną keičiančiai funkcijai.
- Agregato metodai gali būti išreikšti kaip kolekcija funkcijų be būsenos.
- Įvykių saugykla gali būti suvokiama bei naudojama kaip funkcinė duomenų bazė, nes ji perduoda argumentus funkcijoms, kurios keičia agregato būseną. Momentinės kopijos įvykių saugykloje primena įsiminimą atmintyje<sup>15</sup> funkciniam programavime.

### 2.3. Monados

#### 2.3.1. Terminas

Funkciniam programavimui monada yra struktūra, kuri atspindi skaičiavimus, apibrėžtus kaip seka žingsnių. Tipas kartu su monados struktūra apibrėžia ką reiškia vykdyti operacijas viena po kitos bei naudoti to pačio tipo įdėtines funkcijas. Tai leidžia kurti komandų grandines, kurios pažingsniui apdoroja informaciją, kur kiekvienas veiksmas yra dekoruojamas naujomis apdorojimo taisyklėmis, kurias apibrėžia monada [OGS08]. Taip pat monada gali būti laikoma kaip funkcinis projektavimo šablonas, skirtas kurti daugybinius tipus [Mer14].

---

<sup>15</sup><http://en.wikipedia.org/wiki/Memoization>

### 2.3.2. Taisyklės

Anot [Wad95], monadų operacijos turi tenkinti tris taisykles:

- Kairiojo vienetotapatybės (angl. left unit/identity) - suskaičiuojama reikšmė  $a$ , rezultatui priskiriamas  $b$  ir suskaičiuojama  $n$  reikšmė. Rezultatas yra toks pat kaip  $n$  kai  $a$  reikšmė pakeičiama  $b$  reikšme. Parodyta 2 paveikslėlio formulėje.

$$\mathit{unit} \ a \star \lambda b. n = n[a/b].$$

2 pav. Perkėlimo į kairę taisyklė

- dešiniojo vienetotapatybės (angl. right unit/identity) - suskaičiuojama reikšmė  $m$ , rezultatas priskiriamas  $a$  ir grąžinama  $a$ . Rezultatas yra toks pat kaip  $m$ . Parodyta 3 paveikslėlio formulėje.

$$m \star \lambda a. \mathit{unit} \ a = m.$$

3 pav. Perkėlimo į dešinę taisyklė

- Asociatyvumas - suskaičiuojama reikšmė  $m$ , rezultatas priskiriamas  $a$ , suskaičiuojamas  $n$ , rezultatas priskiriamas  $b$ , suskaičiuojama  $o$ . Skliaustelių padėtis šiuo atveju yra nesvarbi. Kintamojo  $a$  apimtis įtraukia  $o$  kairėje, tačiau neįtraukia  $o$  dešinėje, todėl ši taisyklė galioja tik tada kai  $a$  nėra laisvas nuo  $o$ . Parodyta 4 paveikslėlio formulėje.

$$m \star (\lambda a. n \star \lambda b. o) = (m \star \lambda a. n) \star \lambda b. o.$$

4 pav. Asociatyvumas

### 2.3.3. Tolesnis darbas

Kitame mokslinio tyrimo etape ruošiamasi įrodyti, kad funkcinį-reaktyvų programavimą įmanoma taikyti įvykių kaupimo sistemose. Šiam tikslui pasiekti toliau bus nagrinėjamos monados, jų komponavimo būdai. Kad geriau suprasti monadas turbūt prireiks įgauti žinių kategorijų teorijoje, daugiau žinių apie funkcinį programavimą, pavyzdžiui tipizuotos klasės, parametrizuoti tipai ir t.t.

## 2.4. Išvados

Literatūros analizės metu remiantis kitų autorių patirtimi:

- išnagrinėtas funkcinis-reaktyvus programavimas,
- išnagrinėtas įvykių kaupimo principas,
- išnagrinėti įvykių srautai bei operacijos su jais,
- susipažinta su įvykių kaupimu funkciniame programavime,
- susipažinta su monadomis,
- įvaldyta sąvokų sistema, susijusi su nagrinėjama tematika.

### 3. NAUJA!!!!

#### 3.0.1. Random intro

Pagrindinė funkcinio programavimo abstrakcija yra funkcija, kuri priima tam tikrą įvestį, o jos rezultatas yra tam tikrą išvestis. Funkcijos įvestis bei rezultatas gali būti kita funkcija. Funkcinėje programavimo kalboje funkcijos yra pirmos eilės reikšmės.

Priešingai, labiau populiarius imperatyvaus programavimo modelis priima sakinius arba veiksmus kaip pirminius programos konstravimo blokus, kurie modifikuoja būseną. Tokios programos turi nuoseklų kontrolės srautą ir reikalauja samprotavimo apie pašalinius efektus. Iš prigimties jie yra atsparūs kompoziciniam samprotavimui.

Netgi funkcinėse programavimo kalbose, reaktyvios programos yra paprastai parašytos imperatyviu stiliumi, naudojant žemo lygio bei ne paruoštų komponentų abstrakcijas įskaitant atgalinius skambintojus arba objektu paremtais įvykių dorokliais. Tai pririša modelį prie interaktyvumo prie žemo lygio realizacijos detalių tokių kaip laikas bei įvykių dorojimo modeliai.

Funkcinis reaktyvus programavimas reiškia, jog modelis turi išlaikyti funkcinio protamavimo charakteristikas (pavyzdžiui, primityvios kalbos konstrukcijos turi likti pirmosios klasės) įtraukiant reaktyvumą į kalbos modelį. Ypač funkcijos turėtų būti panaikintos, jog operuotų reaktyviomis reikšmėmis ir pačios funkcijos privalo būti reaktyvios.

Pagrindinis FRP tikslas yra suteikti kompozicines ir aukšto lygio abstrakcijas, jog būtų galima kurti reaktyvias programas. Pagrindinės FRP abstrakcijos yra:

- elgesys - laike kintančios reikšmės apibrėžtos kiekviename besitęsiančio laiko taške;
- įvykiai - reikšmės apibrėžtos suskaičiuojamose laiko taškuose.

Funkcinio reaktyvaus programavimo sistema suteiks kombinatorius manipuluoti įvykiais bei elgesiu ir reaguoti į įvykius pakeičiant dalį veikiančios programos kaip atsaką į įvykį. Elgesys bei įvykiai arba tam tikra jų abstrakcija bus pirmos klasės. Funkcinia reaktyvia programavimo kalba realizuotos programos turi būti efektyviai įvykdomos, bet tai yra įrodyta kaip sunkiausias funkcinio reaktyvaus programavimo uždavinys.

Du bendri požiūriai į FRP yra:

- klasikinis FRP - elgesys ir įvykiai yra pirmos klasės ir reaktyvūs objektai;
- signalo-funkcijos FRP - elgesio ir įvykių transformatoriai yra pirmos eilės ir reaktyvūs objektai.

### 3.0.2. Klasikinis FRP

Anksčiausia ir dar vis standartinė FRP formuluotė pateikia du primitivius tipų konstruktorius - Behavior (elgesys) ir Event (įvykis) - kartu su kombinatoriais, kurie pagamina šių tipų reikšmes. Lengviausias semantinis apibrėžimas šiems tipams yra pateiktas 16.

```
type Event a = [(Time, a)]  
type Behavior a = Time -> a
```

Kodo pavyzdys 16 - klasikinio FRP semantiniai tipai

Kai šie du tipų konstruktoriai yra tiesiogiai išreikšti, sistema yra žinoma kaip klasikinė FRP sistema.

### 3.0.3. Klasikinio FRP istorija

Klasikinis FRP buvo originaliai parašytas kaip Fran[8] (Funkcininė reaktyvi animacija). Fran yra karkasas, skirtas deklaratyviai specifikuojant interaktyvias animacijas. Fran atvaizduoja elgesį kaip dvi pavyzdines funkcijas, vieną nuo laiko iki reikšmės ir kitą nuo laiko intervalo (apatinė ir viršutinė laiko riba) iki reikšmės intervalo ir naujo elgesio. Įvykiai atvaizduojami kaip "tobulėjančios reikšmės", kurios imant su laiku pagamina žemesnią laiko ribą kitam atvejui, arba kitą įvykį jeigu jis iš tikrųjų įvyko.

Pirmoji FRP realizacija ne Haskell kalboje buvo Frappe [4], realizuota naudojantis Java Beans karkasu. Frappe yra sukurta remiantis įvykių supratimu bei Beans karkaso susietomis savybėmis (bound properties) teikiant abstrakčias sąsajas FRP įvykiams ir elgesiui bei kombinatorius kaip konkrečias klases, realizuojančias šias sąsajas.

### 3.0.4. Dabartinės klasikinės FRP sistemos

FrTime[1] kalba išplečia Scheme skaičiuoklį nepastoviu priklausomybių grafu, kuris yra sukonstruojamas įvertinus programą. Signalų pasikeitimai atnaušina šį grafą. FrTime nesuteikia atskirų įvykių sąvokų ir pasirenka priklausomybių grafo šakas naudojant sąlyginį elgesio įvertinimą, o ne elgesio pakeitimą, naudojamą FRP sistemų.

Reactive [7] sistema yra dvitaktė (angl. push-pull) funkcinio reaktyvaus programavimo sistema su pirmos klasės elgesiu bei įvykiais. Pirminė Reactive įžvalga yra reaktyvumo (arba kitaip atsako pasikeitimai į įvykius, kurių atsitikimo laikas negalėjo būti žinomas prieš tai) ir laiko priklausomybės atskyrimas. Tai duoda kelių reaktyviajai normalinėj formai, kuri atvaizduoja elgesį kaip konstantą arba paprastai priklausančią nuo laiko reikšmę, kartu su įvykių srauto nešamomis reikšmėmis, kurios irgi yra reaktyvios normalinės formos elgesys. Stūmimu (angl. push) paremtas įverinimas yra pasiekiamas šakojant Haskell gijas, jog būtų įvertintas galvos elgesys kol yra laukiama įvykių srauto įvertinimo. Įvykus įvykiui, dabartinio elgesio gija yra nužudoma ir sukuriamą naują giją įvertinti naują elgesį. Deja Reactive realizacija naudoja nepatvarią techniką, kuri priklauso nuo gijų šakojimosi įvertinant dvi Haskell reikšmes lygiagrečiai, kad būtų galima realizuoti įvykių sujungimą. Tai priklauso nuo bibliotekos autoriaus, kad būtų galima užtikrinti darną kai naudojama ši technika ir priveda prie gijų nuotėkio kai vienas iš sujungtų įvykių yra įvykių sujungimo rezultatas.

Nauja tezė [6] aprašo Elm - autonominių kalbą reaktyvumui. Elm suteikia kombinatorius manipuluoti diskrečiais įvykiais ir kompiliuojasi į Javascript kalbą, kas padaro tai naudinga kliento pusės web programavimui. Tačiau Elm nesuteikia sujungimo arba besitęsiančio laiko elgesio, nors suderinimas yra pateikiamas naudojant diskrečiaus laiko įvykius, kurie yra sužadinami pasikartojančiais intervalais, specifikuotais apibrėžiant įvykį. Tezė teigia, jog Arrowized FRP (signalų-funkcijų FRP) gali būti integruota į Elm, bet suteikia pažai paramos šiam teiginiui.<sup>16</sup>

Reactive-banana [0] biblioteka yra dvitaktė (angl. push-pull) FRP sistema sukurta naudoti su Haskell GUI karkasu. Visų pirma, ji charakterizuoja monadą elgesio ir įvykių kūrimui, kuri gali būti komponuojama bei įvertinama. Ši monada apima

<sup>16</sup>A form of Arrowized FRP employing applicative functors is presented, and justified by the assertion that applicative functors are just arrows with the input type parameter fixed.

konstrukcijas GUI bibliotekos konstrukcijų pririšimui prie primitivių įvykių. Ji privalo būti įkomponuojama į Haskell IO veiksmą įvertinimui įvykti. Reactive-banana realizacija yra panaši į FrTime - naudojant priklausomybių grafą tinklo atnaujinimui įvykus įvykiui. Reactive-banana taip pat kaip Frtime vengia apibendrinto perjungimo vietoje elgesio reikšmių šakojimosi funkcijų, bet išlaiko elgesio ir įvykių atskyrimo. Užuo apibendrinto perjungimo kombinatoriaus, kuris leidžia pakeisti sutartinį elgesį, reactive-banana suteikia žingsninį kombinatorių, kuris pažingsniui sukuria elgesį iš įvykio srauto reikšmių.

### 3.0.5. Signalų funkcijos FRP

Alternatyvus FRP būdas pirmą kartą pasiūlytas darbe apie Fruit[5]. Fruit yra biblioteka, skirta deklaratyviai GUI specifikavimui. Biblioteka naudoja rodyklės[9] sąvoką signalo-funkcijos abstrakcijai. Rodyklės yra abstraktaus tipo konstruktoriai su įvesties ir išvesties tipo parametrais kartu su rinkiniu maršruto parinkimo kombinatorių. Tai demonstruojama 17. Rodyklės idėja Haskell kalboje, įskaitant rodyklių kombinatorių aksiomas, kurias turi tenkinti, yra išvesti iš rodyklių sąvokų iš kategorijų teorijos.

```
(>>>) :: (Arrow a) => a b c -> a c d -> a b d
arr :: (Arrow a) => (b -> c) -> a b c
first :: (Arrow a) => a b c -> a (b, d) (c, d)
second :: (Arrow a) => a b c -> a (d, b) (d, c)
(***) :: (Arrow a) => a b c -> a b d -> a b (c, d)
loop :: (Arrow a) => a (b, d) (c, d) => a b c
```

Kodo pavyzdys 17 - rodyklių kombinatoriai

Signalų funkcijos yra nuo laiko priklausančios ir įvykių bei elgesio reaktyvūs transformatoriai. Elgesys ir įvykiai negali būti tiesiogiai manipuluojami. Šis būdas turi du motyvus: padidina modalumą, kadangi signalo funkcijų įvestis ir išvestis gali būti transformuojama ir tai išvengia didelės laiko klasės ir atminties nuotėkio, kas nutinka kai FRP realizuojamas kaip pirmos klasės elgesys ir įvykiai.

Panašiai kaip ir FrTime, Netwire [17] biblioteka vengia dinaminio perjungimo, šiuo atveju dėl signalo slopinimo. Netwire yra parašyta kaip rodyklės transformatorius. Signalų slopinimas yra įgyvendintas padarant signalo funkcijų išvestį monoidu



ir tada sujungiant signalo funkcijų išvestis. Prislopinta signalo funkcija pagamina monoido nulį (monoid's zero) kaip išvestį. Primityvai apibrėžia elgesio slopinimą ir sukomponuotos signalo funkcijos slopina jeigu jų išvestis dera su monoido nuliu.

Yampa[11] yra rodyklyzuotos FRP sistemos optimizacija, pirmą kartą panaudota Fruit. Yampa realizacija naudoja Generalized Algebraic Datatypes, kad leistų daug didesnę saugaus tipo duomenų tipų klasę signalo funkcijos reprezentacijai. Šis atvaizdavimas kartu su "išmaniais"konstruktoriais ir kombinatoriais suteikia galimybę konstruoti rodyklizuotą FRP sistemą, kuri optimizuoja pati save. Deja pagrindinis neefektyvumas yra nereikalingi įvertinimo žingsniai dėl traukimo paremto (angl. pull-based) įvertinimo. Optimizacija yra speciali ir kiekviena nauja optimizacija reikalauja naujų konstruktorių pridėjimo, taip pat kiekvieno primitivaus kombinatoriaus atnaujinimo kiekvienai konstruktorio kombinacijai palaikyti. Tačiau Yampa parodo aiškų efektyvumo privalumą lyginant su prieš tai aprašytais rodyklizuotomis FRP realizacijomis.

PhD tezė [16] pristatė N-Ary FRP - techniką tipizuojant rodyklizuotas FRP sistemas naudojant priklausomus tipus. Didžioji dalis darbo sudarė priklausomų tipų sistemos korektiškumo įrodymas. Šis darbas pristatė signalų vektorius, tipizuotą konstruktorių, kuris leidžia elgesio bei įvykių atskyrimą FRP sistemos lygyje vietoje įvykių laikymo tik specialiu elgesio tipu.

### 3.0.6. Neįvykdyti iššūkiai

Yra dvi pagrindinės FRP problemos. Pirmą, kol signalo-funkcijos FRP yra iš prigimties saugesnė ir labiau modulinė negu klasikinė FRP, ji dar turi būti efektyviai realizuota. Klasikinės FRP programos yra pažeidžiamos dėl laiko nuotėkio bei priežastingumo pažeidimų dėl galimybės tiesiogiai manipuluoti reaktyviomis reikšmėmis. Antra, sąsaja tarp FRP programų ir daugybės atskirų įvesties ir išvesties šaltinių išlieka specialūs ir daugeliu atveju realizacijos limituotu variantu.

Viena pagrindinė išimtis yra Reactive-banana sistema, kuri suteikia monadą primitivių įvykių konstravimui ir elgesiui iš kuriuos FRP programa gali būti sukonstruota. Tačiau šis būdas yra nelankstus, nes jis reikalauja bibliotekos palaikymo sistemai. Negana to, būnant klasikine FRP sistema, Reactive-banana pritrūksta galimybės transformuoti elgesio bei įvykių įvestį, kadangi visa įvestis yra neišreikštinė.

### 3.0.7. Naudota čia

- [0] Heinrich Apfelmus. reactive-banana library. <http://hackage.haskell.org/package/reactive-banana>.
- [1] <http://cs.brown.edu/people/sk/Publications/Papers/Published/ick-adapt-oo-fwk-frp/paper.pdf>
- [4] Antony Courtney. Frapp 'e: Functional reactive programming in Java. In Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages, PADL '01, pages 29–44, London, UK, UK, 2001. Springer-Verlag.
- [5] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In Proceedings of the 2001 Haskell Workshop, pages 41–69, 2001.
- [6] Evan Czaplicki. Elm: Concurrent FRP for functional GUIs. <http://www.testblogpleaseignorescontent/uploads/2012/03/thesis.pdf>, 2012.
- [7] Conal Elliott. Push-pull functional reactive programming. In Haskell Symposium, 2009.]
- [8] Conal Elliott and Paul Hudak. Functional reactive animation. In Proceedings of the second ACM SIGPLAN international conference on Functional programming, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.
- [9] John Hughes. Generalising monads to arrows. Science of Computer Programming, 37(13):67 – 111, 2000
- [11] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, ICFP '05, pages 54–65, New York, NY, USA, 2005. ACM.
- [16] Neil Schulthorpe. Towards Safe and Efficient Functional Reactive Programming. PhD thesis, University of Nottingham, UK, 2011.
- [17] Ertugrul Söylemez. netwirelibrary. <http://hackage.haskell.org/package/netwire-3.1.0>.

### 3.0.8. Fran Abstract

Fran (Functional Reactive Animation) yra kolekcija duomenų tipų ir funkcijų, skirtų komponuoti labai interaktyvias multimedijos animacijas. Pagrindinės Fran idėjos yra elgesio ir įvykio sąvokos. Elgesys yra laike kintančios reaktyvios reikš-

mės, o įvykiai yra rinkinys sudėtingų būsenų galimai nešančių gausų informacijos kieki. Dauguma tradicinių reikšmių gali būti laikomos elgesiu. Kai vaizdai yra apdorojami - jie tampa animacijomis. Nors šios didėjos yra laikomos kaip duomenų tipai, o ne programavimo kalba, jiems galima suteikti semantiką įskaitant teisingą apdorojimą realiu laiku, jog būtų galima apgalvoti realizaciją. Metodas efektyviai aptikti įvykius naudojantis intervalų analizę yra taipogi apibūdintas. Jis remiasi daline įvykių laikų domeno srities informacijos struktūra. Fran buvo realizuotas Hugs suteikdamas stebėtinai gerą našumą interpretatoriumi paremtose sistemose. Keli pavyzdžiai yra duoti, įskaitant galimybę apibūdinti fizinius reiškinius įskaitant gravitaciją, svyruokles, greitį, pagreitį ir t.t. naudojantis paprastas diferencialines lygtis.

### **3.0.9. Įvadas Fran**

Interaktyvios multimedijos animacijų kūrimas (įskaitant audio, nuotraukas, video, 2D ir 3D grafiką) ilgai buvo sudėtingas ir nuobodus procesas. Tikima, jog sunkumas kyla dėl pakankamai aukšto lygio abstrakcijų nebuvimo, ir ypač dėl sunkumo atskirti modeliavimo ir prezentacijos lygmenis arba kitais žodžiais, tarp to kas yra animacija ir kaip ji turėtų būti atvaizduota. Dėl šios priežasties, programos turi išreikštinai valdyti bendrus realizacijos detales, kurios neturi nieko bendro su animacijos turiniu, o ne patį atvaizdavimą naudojantis žemo lygio vaizduoklio bibliotekas. Šios realizacijos detalės apima:

- modeliavimą ir kadro generavimą pažingsniui keliaujant laiku nepaisant to, jog animaciją yra iš esmės tolydi;
- judesio įvesties įvykių sekų surinkimą ir apdorojimą nepaisant to, jog judesio įvestis iš esmės yra tolydi;
- laiko dalijimą atnaujinant kiekvieną laiką besikeičiančią animacijos parametą nepaisant to, jog šie parametrai iš esmės lygiagrečiai skiriasi.

Leidžiant programuotojams išreikšti interaktyvios informacijos "kas", kažkas gali tikėtis automatizuoti prezentacijos "kaip". Šiuo požiūriu, neturėtų būti netikėta, jog rinkinys išraiškingų rekursyvių duomenų tipų sujungtų su deklaratyvia programavimo kalba leidžia patogiai modeliuoti animacijas, priešingai nei bendrinė praktika

naudoti imperatyvias kalbas sutartinai mišriam modeliavimo/prezentacijos stiliui. Taipogi yra rasta ne griežta semantika, aukštesnės eilės funkcijos, stiprus polimorfizmas tipizavimas ir sistemingas perkrovimas yra vertingos kalpos savybės, leidžiančios palaikyti sumodeliuotą animaciją. Dėl šių priežasčių, Fran suteikia duomenų tipus programavimo kalboje Haskell.

Modeliavimo privalumai lyginant su prezentacija

Modeliavimo privalumai prieš animaciją yra panašūs į funkcinės (arba galima sakyti deklaratyvios) programavimo kalbos paradigmą ir apima aiškumą, kūrimo lengvumą, komponavimą ir šviesią semantiką. Be šių yra programai būdingų privalumų, tam tikrais atvejais patrauklesnių iš programinės įrangos kūrėjos bei galutinio vartotojo perspektyvos. Šie privalumai apima:

- Kūrimas - turinio kūrimo sistemos natūraliai konstruoja modelius, nes šių sistemų galutinis vartotojas mąsto modelio terminais ir paprasta neturi nei noro nei patirties programavimo prezentacijos detalėse.
- Optimizuojamumas - modeliu paremtos sistemos turi prezentacijos subsystemą, kuri gali atvaizduoti bet koki modelį, kuris gali būti sukurtas sistemoje. Egzistuoja daug galimybių optimizacijai, nes aukšto lygio informacijos detalės yra prieinamos prezentacijos subsystemai.
- Reguliavimas - prezentacijos subsystema gali lengviau apibrėžti detalių išsamumo lygio valdymą bei pavyzdžių ėmimo dažnį, būtiną interaktyvioms animacijoms, remiantis reginio sudėtingumo, mašinos greičiu ir apkrova ir t.t.
- Mobilumas ir saugumas - modeliavimo platformos nepriklausomumas palengvina mobilių aplikacijų, kurios yra įrodytai saugios WWW(World Wide Web) programos, konstravimą.

Modeliavimo esmė

Yra keturios pagrindinės modeliavimo idėjos:

- Laikinas modeliavimas. Reikšmės, vadinamos elgesiu, kurios kinta bėgant laikui yra labiausiai dominančios. Elgesys yra pirmos klasės reikšmės ir sukurtos kompoziciškai. Lygiagretumas yra išreikštas natūraliai ir neišreikštinai. Pavyzdžiui, sekanti išraiška išreiškia animaciją (paveikslėlio elgesį), kas yra ap-

skritimas ant kvadrato. Laiko taške  $t$ , apskritimas turi dydį  $\sin t$  ir kvadratas turi dydį  $\cos t$ .

$\text{bigger}(\sin \text{time}) \text{ circle } 'over' \text{ bigger}(\cos \text{time}) \text{ square}$

- Įvykių modeliavimas. Kaip ir elgesys, įvykiai yra pirmos eilės reikšmės. Įvykiai gali reikšti tam tikrus nutikimus realiame pasaulyje (pavyzdžiui, pelės mygtuko paspaudimas) arba predikatus paremtus animacijos parametrais (pavyzdžiui, artimumą arba susidūrimą). Tokie įvykiai gali būti sujungti su kitais iki norimo sudėtingumo taip atskiriant sudėtingą animacijos logiką į semantiškai turiningus, moduliūs konstravimo blokus. Pavyzdžiui, įvykis, aprašantis pirmą kairio mygtuko paspaudimą po laiko  $t_0$  yra paprasčiausiai  $1bp\ t_0$ ; aprašantis laiko kvadratą lygų penkiems yra *predicate*  $(pow(time, 2) == 5\ t_0)$  ir jų loginė disjunkcija  $1bp\ t_0 \ ./.\ predicate\ (pow(time, 2) == 5)\ t_0$
- Deklaratyvus reaktyvumas. Elgesys dažnai yra natūraliai išreiškiamas kaip atsakas į įvykį. Bet netgi šis reaktyvus elgesys turi deklaratyvią semantiką dėl būsenos pasikeitimų, neretai įtraukiamų į įvykiais paremtą formalizmą. Pavyzdžiui, spalvos reikšmės elgesys, kuris periodiškai keičiasi iš raudonos į žalią su kiekvienu mygtuko paspaudimu gali būti aprašytas kaip paprastas pasikartojimas:

```

colorCycle t0 =
  red 'untilB' 1bp t0 *==> \\t1 ->
  green 'untilB' 1bp t0 *==> \\t1 ->
  colorCycle t2

```

- Polimorfinė medija. Laike besikeičiančių medijų (nuotraukos, video, garsas, 3D grafika) įvairovė ir šių tipų parametrai (erdvinės transformacijos, spalvos, taškai, vektoriai, skaičiai) turi savo pačių specialiai tipui opecijas (pavyzdžiui, nuotraukų sukimas, garso maišymas, skaitmeninė sudėtis), tačiau sutelpa į bendrinį elgesio ir reaktyvumo karkasą. Pavyzdžiui, 'untilB' operaciją naudojama prieš tai yra polimorfinė, tinkanti bet kuriai laike besikeičiančiai reikšmei.

### 3.0.10. Formali semantika

### 3.0.11. Semantinės domeno sritys

Abstrakti laiko domeno sritis yra vadinama *Time*. Abstrakti polimorfio elgesio ( $\alpha$ -behaviors) domeno sritis yra žymima  $Behavior_\alpha$ , o polimorfinių įvykių ( $\alpha$ -events) yra žymima  $Event_\alpha$ .

Daugiausia šių sričių (sveikieji skaičiai, loginės reikšmės) yra standartinės ir nereikalauja paaiškinimo. *Time* domeno sritis reikalauja specialaus traktavimo kadangi laiko reikšmės įtraukia dalinius elementus. Ypač yra žinoma, jog laikas bent jau kažkokia reikšmė netgi nežinant galutinės reikšmės. Tiksliau laiko domeno sritį galima apibrėžti taip: tarkime  $R$  yra rinkinys realių skaičių ir jame yra elementai  $\infty$  ir  $-\infty$ . Šis rinkinys turi standartinį aritmetinį rikiavimą  $\leq$  įskaitant faktą, jog  $-\infty \leq \infty$  kiekvienam  $x \in R$ .

Dabar apibrėžkime laiką kaip  $Time = R + R$ , kur elementai antrame rinkinyje  $R$  yra atskiriami rikiaviu  $\geq$  (pavyzdžiui,  $\geq 42$  skaitytume kaip "daugiau arba lygu 42". Tada galima apibrėžti  $\perp Time = \geq(-\infty)$  ir domeno srities (pavyzdžiui, informacijos) rikiavimą pagal laiką:

$$x \sqsubseteq x, \forall x \in R$$

$$\geq x \sqsubseteq y \text{ if } x \leq y, \forall x, y \in R$$

$$\geq x \sqsubseteq \geq y \text{ if } x \leq y, \forall x, y \in R$$

Lengva pastebėti, jog  $\perp Time$  yra apatinis elementas. Svarbu paminėti, jog  $y$  yra tik bent viršutinė dalinių elementų rinkinio riba (angl. least upper bound), kuri apytiksliai lygi:

$$y = \sqcup \{ \geq \mid x \leq y \}$$

Kadangi laiko domeno rikiavimas yra grandinės tipo ir kiekviena tokia grandinė turi bent viršutinę ribą (prisiminkime  $R$  turi viršutinį elementą  $\infty$ ), laiko domenai

yra pilnos dalinės tvarkos (angl. complete partial order). Šis faktas yra būtinas, jog būtų galima užtikrinti, kad rekursyvūs apibrėžimai yra gerai apibrėžti.

*Time* rinkinio elementai naudingiausi įvertinant laiką kada atsitinka įvykis. Įvykis kurio laikas apytiksliai  $\geq t$  yra tas, kurio konkretus įvykimo laikas yra didesnis nei  $t$ . Svarbu, jog įvykio, kuris niekada neįvyksta, laikas yra  $\infty$ , bent viršutinė  $R$  riba.

Galiausiai apibrėžimą galima išplėsti aritmetiniui operatoriui  $\leq$  visam *Time* apibrėžiant jo elgesį visuose subdomenuose:

$$x \leq_{\geq} y \text{ if } x \leq y$$

Tai gali būti skaitoma kaip: "Laikas  $x$  yra mažesnis arba lygus laikui, kuris yra bent  $y$  jeigu  $x$  mažiau arba lygu  $y$ ". Lengva parodyti, jog ši išplėsta tipo  $Time \rightarrow Time \rightarrow Bool$  funkcija yra tolydi atsižvelgiant į  $\sqsubseteq$ .

### 3.0.12. Semantinės funkcijos

Polimornio elgesio interpretaciją galima apibrėžti kaip funkciją, kuri priima polimorfinę reikšmę bei pagamina elgesio  $b$  reikšmę laiku  $t$ .

$$at : Behavior_{\alpha} \rightarrow Time \rightarrow \alpha$$

Tada galima apibrėžti polimorfinių įvykių interpretaciją kaip paprastą ir negriežtą  $Time \times \alpha$  porą, aprašančią laiką ir informaciją, susijusią su įvykio atsitikimu:

$$occ : Event_{\alpha} \rightarrow Time \times \alpha$$

Žinant semantinę domeno sritį, galima pateikti formalias įvykių kombinatorių interpretacijas.

### 3.0.13. Elgesio semantika

Elgesys yra kuriamas iš kito elgesio, statinių (nesikeičiančių laike) reikšmių ir įvykių naudojantis kolekcija konstruktorių (kombinatorių).

- **Laikas.** Paprasčiausias primityvus elgesys yra laikas - time, kurio semantika yra:

$$time : Behavior_{Time}$$

$$\mathbf{at}[[time]]t = t$$

Šiuo atveju  $\mathbf{at}[[time]]t = t$  yra tik *Time* tapatumo funkcija.

-



## 4. Sąvokų apibrėžimai

- **Agregatas** (angl. aggregate) - DDD modelis, rinkinys domeno objektų, kurie gali būti laikomi kaip visuma.
- **Derinimas** (angl. debugging) - riktų bei klaidų paieška programinėje įrangoje bei jų taisymas.
- **Esybė** (angl. entity) - kažkas, kas egzistuoja pats savaime, faktiškai arba hipotetiškai.
- **Horizontalus išplečiamumas** (angl. horizontal scaling) – galimybė sujungti daugybę techninės ar programinės įrangos esybių taip, jog jos dirbtų kaip visuma. Pavyzdžiui, galima pridėti keletą serverių pasinaudojant grupavimu arba apkrovos paskirstymu taip pagerinant sistemos našumą bei prieinamumą.
- **Metaduomenys** (angl. metadata) - duomenys apie kitus duomenis.

## 5. Santrumpos ir paaiškinimai

- **CQRS** (angl. Command Query Responsibility Segregation) – komandų užklausų atsakomybių atskyrimas.
- **DDD** (angl. Domain-Driven Design) – būdas kurti programinę įrangą, skirtą spręsti sudėtingus uždavinius, bei apjungti realizaciją kartu su augančiu domeno modeliu.
- **NoSQL** – duomenų bazė, skirta architektūriniais modeliams, kuriems nereikia palaikyti stiprios darnos principo, kuris naudojamas reliacinėse duomenų bazėse. Tai įgalina horizontalų išplečiamumą bei aukštesnį prieinamumą.
- **ORM** (angl. Object-Relational Mapping) – programavimo technika duomenų konvertavimui tarp nesuderinamų sistemų tipų, naudojama objektinio programavimo kalbose. Pavyzdys: JAVA programavimo kalba bei Oracle duomenų bazė.

## Literatūros sąrašas

- [Ams] Edward Amsden. A survey of functional reactive programming. *Rochester Institute of Technology*.
- [Bas] Tim Bass. Mythbusters: Event Stream Processing Versus Complex Event Processing. page 1. Invited talk.
- [BDM<sup>+</sup>13] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. Microsoft patterns & practices, 1st edition, 2013.
- [Cza12] Evan Czaplicki. Elm: Concurrent frp for functional guis. Master’s thesis, 30 March 2012.
- [EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.
- [Fit12] Sean Fitzgerald. A pattern for state machine persistence using event sourcing, cqrs and a visual workbench. Master’s thesis, 21 March 2012.
- [KW92] David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 134–143. Springer, 1992.
- [Mer14] L.G. Meredith. *Pro Scala: Monadic Design Patterns for the Web*. Artima, 2014.
- [MO12] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.
- [NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell ’02, pages 51–64, New York, NY, USA, 2002. ACM.
- [OGS08] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O’Reilly Media, Inc., 1st edition, 2008.

- [OSG04] OSGi Alliance. Listener Pattern Considered Harmful: The "Whiteboard"Pattern. Technical whitepaper, OSGi, 2004.
- [Ver13] V. Vernon. *Implementing Domain-Driven Design*. Pearson Education, 2013.
- [Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.
- [You10] Greg Young. Cqrs documents by greg young. Technical whitepaper, 2010.