TARTU ÜLIKOOL

Arvutiteaduse instituut

Informaatika õppekava

Peter Kallaste

Efektisüsteemide õpetamine Haskellis

Bakalaurusetöö (9 EAP)

Juhendaja: Kalmer Apinis

Tartu 2021

Efektisüsteemide õpetamine Haskellis

Lühikokkuvõte:

Käesolevas bakalaureusetöös loodi õppematerjalid Haskelli efektisüsteemide kohta. Selle töö käigus vaadeldi nelja erinevat efektisüsteemi (freer-simple, fused-effects, polysemy ja effet) ja nende seast valiti üks, mille kohta moodustati õppematerjalid. Võrdluse käigus valiti välja fused-effects teek ja moodustati sellest õppematerjale. Sellega seoses räägiti efektide kasutamisest funktsioonides ja õpetati neid käivitama programmides. Veel näidati uute efektide loomist, et tekkis arusaam efektisüsteemide kasulikkusest. Õppematerjali näidete juurde koostati ülesanded, mille läbimisel tudeng peaks oskama kirjutada lihtsaid efekte kasutatavaid meetodeid ja rakendada neid oma projektides.

Võtmesõnad:

Haskell, efektisüsteem, efektide loomine, monaad, monaaditeisendaja, õppematerjal

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimis-teooria)

Study materials for Haskell effect systems

Abstract:

In this thesis, a study material is created for Haskell effect systems. This thesis contains analyses of four different effect systems (freer-simple, fused-effects, polysemy and effet) where the fused-effects package is chosen for creating study materials. Study materials include information about how to use effects in methods and how to run those effects. One of the essential parts of effect systems is creating new effects, and with that more profound understanding of effect systems is developed. Study materials also contain exercises that help along the way. With that, the student should understand how to make a simple effect and use it in their projects.

Keywords:

Haskell, effect system, defining effects, monad, monad transformer, study material

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

Sisukord

Sissejuhatus

Kasulikud on need programmid, mis midagi teevad ja tavaliselt selle töö käigus juhtuvad mingid efektid. Isegi lühikestes programmides võib olla mitmeid efekte, näiteks teksti sisestamine ja väljatrükk, juhuarvude genereerimine, andmete lugemine ja suhtlus üle võrgu. Nende efektide haldamiseks on Haskellis võimalik kasutada lisasid, mis töötavad efektisüsteemi põhimõttel. Kuid nende kasutuse kohta puudub eestikeelseid materjale ja enamus ingliskeelsed materjalid pole juba algajatele mõeldud.

Käesoleva bakalaureusetöö eesmärgiks on uurida erinevaid efektisüsteeme Haskellis ja ühe efektisüsteemi põhjal teha õppematerjale, mida saab kasutada näiteks "Programmeerimis-keelte" kursusel või teises aines lisamaterjalina. Antud töös on valitud neli erinevat efektisüsteemi ja moodustatud nendest lühikirjeldused. Nende tutvustuste käigus vaadatakse nende tunnusjooni ja võrreldakse neid teiste efektisüsteemidega. Selle protsessi käigus valiti välja fused-effectsi teek, mille kohta moodustati lõputöö eesmärgiks õppematerjalid.

Esimeses peatükis tutvustatakse Tartu Ülikooli kursust "Programmeerimiskeeled". Kirjeldatakse selle eesmärke, aine sisu ja vaadatakse selle ülesehitust. Teises peatükis kirjeldatakse funktsionaalset programmeerimist ja vaadatakse Haskelli keeles sissetoodud monaade ning kuidas need efektisüsteemidega seotud. Selle käigus tutvustatakse erinevaid meetodeid monaadide kokku panemiseks. Kolmandas peatükis on valitud neli efektisüsteemi (freer-simple, fused-effects, polysemy, effet) ja nende põhjal tehtud kirjeldused, mis tutvustavad nende ülesehitust ja võrreldakse neid omavahel. Neljandas peatükis räägitakse õppematerjalide koostamisest ja tuuakse välja, miks fused-effectsi teek on valitud materjalide jaoks.

1. Kursuse “Programmeerimiskeeled” kirjeldus

“Programmeerimiskeeled” (MTAT.03.006) on 6 EAP mahuline kursus. Selle aine eesmärgiks on õpetada tudengitele teisi programmeerimiskeeli, mis oleksid alternatiivid Java ja Pythoni keeltele [1]. Peamiselt vaadatakse funktsionaalse programmeerimise paradigmat kasutades Haskell ja Scala keeli. Selle kaudu tekib imperatiivse keele asemel teine mõtteviis ja avardub arusaam teistest programmeerimiskeeltest. Laialdaselt vaadatakse funktsionaalse programmeerimise tüüpilisi tunnuseid ja õpitakse neid kasutama enda loodud programmides. Kursuse esimesel poolel õpetatakse Haskelli keelt, mis on puhas funktsionaalne keel ja teisel poolel Scala keelt, mis on funktsionaalse ja imperatiivse keele segu. Iga nädalal toimub üks loeng ja üks praktikum ehk kokku toimub 16 loengut ja 16 praktikumi. Antud õppematerjal oleks selle aine ühe praktikumi pikkune.

2. Ülesande püstitus

Selle lõputöö eesmärgiks on luua õppematerjal efektisüsteemidest. Selleks, et saada aga aru, mis need on ja mis probleeme nad lahendavad, tuleb alustada kaugemalt.

2.1 Funktsionaalne programmeerimine

Funktsionaalne programmeerimine on üks programmeerimise paradigmadest, kus enamus asju vaadeldakse rohkem matemaatiliste objektidena [2]. Kui imperatiivses programmeerimises vaadatakse koodi ridu ükshaaval ja täidetakse neid järjest, siis funktsionaalse programmeerimises on rohkem oluline funktsioonide omavaheline seos ja täidetakse neid nii kuidas on kõige parem. Funktsioonid võivad käituda nagu andmed, kus üks funktsioon võib olla teise funktsiooni argumendiks. Funktsionaalse keele üheks peamiseks punktiks on see, et sama sisendi korral on funktsiooni väljund alati sama. Neid nimetatakse puhtateks funktsioonideks. Imperatiivses keeles võib sama sisendi puhul funktsioon oma väärtusi muuta ja tulemus võib olla teine. Sellist käitumist nimetatakse kõrvalefektiks. Funktsionaalses keeles püütakse vältida sellist olukorda. Veel üheks plussiks on selle väljendusvõime ja paindlikkus igasuguste programmide koostamiseks ning programmi suurenemisel on koodist kergem aru saada võrreldes imperatiivse keelega. Üheks miinuseks on see, et funktsionaalne programmeerimiskeel on oma ülesehituse tõttu ressursimahukam ja võrreldes mõnede teiste keeltega aeglasem. Üldiselt on funktsionaalne programmeerimiskeel suhteliselt halb igasuguste interaktiivsete asjade jaoks, näiteks ekraanile trükkimine, kõvaketta pealt lugemine ja internetiga ühendamine [2]. Selle jaoks on vaja kasutada erilisi funktsioone, mis võimaldavad teha kõrvalefekte. Kuigi

puhas funktsionaalne keel eeldab, et kõik toimub mingi kindla keskkonna sees ja puuduvad kõrvalefektid.

2.1 Haskelli lühitutvustus

Haskell on üks populaarsemaid programmeerimiskeeli, mis põhineb puhtal funktsionaalsel programmeerimisel ehk igasugused kõrvalefektid puuduvad [3]. Haskelli põhiline omadus on see, et kõik andmed on tugevalt tüübitud. Seda võimaldab hea tüübiklasside süsteem. Kõikidel objektidel on omad tüübid ja need kõik on staatiliselt tüübitud ehk programmi avaldise tüübid on kompileerimise ajaks teada ja need ei muutu. Üheks Haskelli keele iseloomujooneks on laisk väärtustamine ehk väärtustatakse ainult need andmed, mida tegelikult vaja läheb ja alles siis kui neid tegelikult vaja on. See tingimus võimaldab defineerida lõpmatud järjendid ja nende peal sooritada matemaatilisi operatsioone. Haskelli keeles on koodi täiustamiseks võimalik kasutada erinevaid teeke (ingl *library*), mis toovad programmi sisse teisi funktsioone ja tüüpe. Neid kasutades on võimalik oma programmi kergesti tuua sisse efektisüsteeme.

2.2 Monaaditeisendajad

Kui minna sügavamale Haskelli keskkonda ja rohkem selle lõputöö kohasemale teemale, siis tuleks rääkida algul monaadidest. Enamus programmid kasutavad mingit tüüpi efekte, näiteks ekraanile trükkimine, teksti sisestamine ja võrguga suhtlemine. Kuna need funktsioonid ei saa kuidagi olla puhtad, siis saab nende defineerimisel kasutada monaade. Kõrvalefektide sissetoomiseks on veel teisigi lahendusi, aga Haskellis otsustati lisada monaadid. Monaadid on Haskellis ühe muutujaga tüübipere, mille jaoks on defineeritud erinevad meetodid [4]. Üldiselt võib neid vaadata nagu konteinereid, milles saab hoida mingit tüüpi andmeid ja mille peal saab tööd teha. Monaadid on ühed peamised vahendid Haskellis, millega saab modelleerida puhaste ja mittepuhaste arvutuste kombineerimist. Haskelli peameetodi tagastus tüübiks on IO monaad ja see on ainuke koht, kus võivad tekkida kõrvalefektid, mida puhtal funktsionaalkeelel ei tohiks olla [3]. Tegelikult on Haskell ikkagi teoreetiliselt puhas keel ja kõrvalefektide soodustamiseks on kasutatud kavalaid lahendusi. Näitena võib tuua funktsioon, mis peab lugema mingist sisendist arvusid, salvestama need ühte muutajasse (mis pärast oleks teistele programmidele kättesaadav) ja kui arvude hulk on väiksem kui nõutud, siis funktsioon saadab kasutajale veateate. Haskelli süsteem tagab selle, et selline kõrvalefekti süsteemis saaks toimuda ja seetõttu saab programm olla rohkem interaktiivsem.

Õppimise eesmärgil on monaade päris kasulik vaadata ükshaaval, kuid reaalelu programmides on palju kasulikum mitmed monaadid kokku panna ja seetõttu leiutati sellised objektid nagu

monaaditesisendajad (ingl *monad transformer*). Monaaditeisndajad on lihtsad objektid, mis saavad enda sees hoida monaade või teisi monaaditeisendajad [5]. Nende vahel liikumiseks on tavaliselt loodud *lift* ja *return* funktsioonid, kus tuleks tähele panna, et *return* funktsioon pole sama, mis tüüpiliselt esineb imperatiivses programmeerimis keeles. Haskelli keeles lisab monaaditeisesendajad transformer teek, mis sisaldab lihtsaid tüüpilisi funktsioone monaadide kokkupanemiseks ja veel MTL (*Monad Transformer Library*) teek, mis sisemiselt kasutab transformer teeki ja lisab mõningaid lihtsustusi monaadide sidumiseks ja lahendab mõningaid probleeme, mis leidub transformer teegis [3]. Näiteks, üheks kõige tüüpilisemaks probleemiks on mitmekihilise monaaditeisendaja defineerimine ja selle tulemusel peab nende vahel liikumiseks kasutama korduvaid *lift* ja *return* operatsioone, mis võib muutuda väga tüütuks.

2.3 MTL

Nagu ennegi juba öeldud, siis MTL (*Monad Transformer Library*) [6] on üks kõige populaarsemaid monaaditeisendajate teeke ja mille peamiseks funktsiooniks on teha lihtsamaks töötlemine pesastatud monaaditeisendajatega ning seda teeki kasutades on võimalik veel luua uusi monaaditeisendajaid.

Kuna MTL on mõeldud monaaditeisendajatega töötamiseks, siis tihti on vaja tõsta üks monaad teiseks monaaditeisendaja tüübiks ja selle sooritamiseks on MTL teegis olemas *lift* nimega meetod (`lift :: (Monad m, MonadTrans t) => m a -> t m a`), kus argumendiks on *m a* tüüpi monaad ja see tõstetakse *t* tüüpi monaaditeisendajaks [6].

MTL on saanud inspiratsiooni Mark P. Jonesi uurimistöös välja toodud monaadidest ja sellepärast on nende monaadide funktsioonid väga sarnased töös esitatud monaadidega [4, 6]. MTL koosneb järgmistest klassidest:

- `Control.Monad.Cont` – jätkamise monaadi kasutatakse ühe funktsiooni tulemuse edastamiseks teise funktsiooni argumentidesse. Selle tulemusel saab sooritada keerukaid operatsioone funktsiooni käigus.
- `Control.Monad.Expect` – vea tagastamiseks loodud monaad (*Error* monaadi asemel).
- `Control.Monad.List` – järjendi monaad kombineerib ühe järjendi monaadi mingi teise monaadiga.
- `Control.Monad.Trans` – suudab panna ühe monaadi teise sisse.
- `Control.Monad.Reader` – antud monaaditeisendaja saab lugeda ühise keskkonna väärtusi ja edastada neid teisetele funktsioonidele.

- `Control.Monad.Writer` – suudab väärtusi väljastada kui ka mingit väärtust tagastada.
- `Control.Monad.State` – antud monaaditeisendaja suudab lugeda ja muuta oma keskkonna seisundit.
- `Control.Monad.RWS` – kombineerib *Reader*, *Writer* ja *State* monaaditeisendajaid.
- `Control.Monad.Identity` – identiteedi monaad seob ühe funktsiooni argumentide külge teise funktsiooni.

Üks peamine MTL teegi probleeme tuleb välja uute monaaditeisendajate kirjutamisel, sest kohandatud teisendajate tegemisel peab kirjutama väga palju sarnast koodi lihtsate asjade lisamiseks [4]. Seda nimetatakse $O(n^2)$ isendi probleemiks.

```
instance MonadReader r m => MonadReader r (ExceptT e m) where
    ask   = lift ask
    local = mapExceptT . local
    reader = lift . reader
instance MonadReader r m => MonadReader r (IdentityT m) where
    ask   = lift ask
    local = mapIdentityT . local
    reader = lift . reader
-- sama ContT, MaybeT, StateT, WriterT jaoks
```

Joonis 1. MTL koodis MonadReader tüübi koostamine [7]

Eelnev kooditükk on välja toodud MTL enda koodist ja kus on näha, et uue *MonadReader* tüübi loomise jaoks peab veel eraldi defineerima sidusused teiste monaadidega ning selles esinebki $O(n^2)$ isendi probleemi. Kui oma teisendajaid mitte kirjutada, siis sarnaseid probleeme enam ei esine, aga kood võib minna segaseks ja ainult ühe programmi põhiseks [3].

Kuigi Dominic Orchard [8] on esile toonud, et monaadid pole väga detailsed ehk, et me saame vaadata selle seisundit, aga sisemise süsteemi efekte pole võimalik jälgida, ja kõige olulisemalt monaade pole kerge lugeda ning vajavaid lisakommentaare koodis, et nendest täielikult aru saada. Eelnev uurimus tõi välja, et nende vigade vastu on loodud erinevaid efektisüsteeme, mis suuremas osas aitavad luua monaade efektisüsteemi põhiselt. Nendest puudustest vaatamata on Figueroa [9] oma uuringus välja toonud, et monaadid on laialdaselt kasutuses ja 30,8% Haskelli pakettidest kasutavad MTL teeki, kõige rohkem kasutatakse sellest seisundi monaadi. Figueroa lisab veel huvitava fakti, et teised monaadide teegid võivad olla loodud suuremas osas uurimistöö eesmärgil ja leiab vähest kasutust reaalelu programmides.

2.4 Efektisüsteemid

Monaaditeisendajad on väga populaarsed Haskelli süsteemides, aga eelnevaid puudusi arvestades on nad ikkagi väga limiteeritud. Selle parandamiseks on välja mõeldud efektisüsteemid, mille funktsioon on väga sarnane monaaditeisendajatega. Need on loodud selleks, et mitu monaadi kokku panna ja programm luua rohkem interaktiivsemaks. Efektisüsteemi sisemuses defineeritakse algul kõik efektid ära ja pärast saadetakse need ükshaaval käitlejatele (ingl *handler*) ja programm hakkab ootama nendelt vastuseid [10]. Vastuseks võib olla mingi tagastusväärtus või veateade. Kui programmis pole defineeritud mingi kindel järjekord nendele efektidele, siis käitleja käivitab need efektid esinemise järjekorras ja selle tõttu saab efektisüsteemides efekte defineerida samuti suvaliselt. Eelnev on veel üks probleem algsete monaadideteisendajatega ehk kui on defineeritud *State (Except)* või *Except (State)*, siis süsteem hakkab töötama teistmoodi ning mingis olukorras võime saada teisi vastuseid [3]. Veel üheks heaks asjaks on see, et efektisüsteemide põhimõttel hoitakse kõik efektid mingi ühe kindla monaadi sees ja see elimineerib $O(n^2)$ isendi probleemi, mis esineb monaaditeisendajatega.

3. Efektisüsteemide teegid

Töö üheks ülesandeks on vaadata nelja kõige populaarsemaid efektisüsteeme ning nende seast võtta üks efektisüsteem, mida kasutada õppematerjalide koostamiseks.

3.1 freer-simple

Freer-simple on Alexis King poolt loodud efektisüsteemi teek, mis on hargmik freer-effects teegist, mis on omakorda hargmik freer teegist. Need kõik on loodud Oleg Kiselyovi „Freer Monads, More Extensible Effects“ uurimistöö põhjal olevast materjalist [11].

Freer-simple on üks Haskelli efektide teekidest, mis teeb lihtsamaks erinevate efektide jälgimise ja nende rakendamist süsteemidesse [11]. Selleks võib olla näiteks failisüsteemis liikumine, juhuslikke numbrite loomine, erandi viskamine ja globaalsete väärtuste muutmine. Freer-simple dokumentatsioonis tuuakse välja, et selle peamisteks tunnusteks on tõhusa efektisüsteemi loomine, põhimonaadide rakendused (*Error*, *Reader*, *State*, *Trace*, *Writer* monaadid) ja lihtne moodus oma efektide loomiseks [11].

Freer-simple süsteemis on defineeritud monaad nimega `Eff r a`, milles *r* nimega parameetris hoitakse efektide nimekirja ja *a* on selle monaadi tagastustüübiks. *Eff* monaad on konteiner, kus hoitakse kõiki neid efekte ja pärast saab neid sealt ükshaaval tööle panna. Näiteks kui on vaja seisundi, lugeja ja IO efekte kasutada, siis saab meetodi tüübiks panna: `Eff '[State String, Reader Int, IO] Bool` ja `Bool` oleks selles definitsioonis tagastustüübiks [12]. Selle põhimõtte pärast on freer-simple teegis lihtne monaade kokku panna ja oma efektide loomise tõttu suureneb programmi koodi arusaadavus võrreldes tavalise monaadide kasutusega.

Kõik järgnevad koodi näited on inspireeritud fused-effects dokumentatsioonis toodud näidetest [13] ja kohandatud selliselt, et töötaks konkreetses Haskelli teegis. Sellise tulemusega on neid pärast lihtsam võrrelda ja vaadata nende süntaktilisi erinevusi.

Freer-simple teegis saab efekte töötlevate meetodite defineerimiseks kasutada *Members* nimega tüübipiirajat `(Member (eff :: Type -> Type) effs)` või *DataKinds* laiendit, milles kõik efektid lisatakse ühte järjendisse [11]. Järgnevas näites kasutatakse erinevaid meetodeid efektide defineerimiseks.

```haskell
naide1 :: Member (State String) effs => Eff effs ()
naide1 = get >>= \ s -> put ("Tere, " ++ s)

-- Kasutades: {-# LANGUAGE DataKinds #-}
naide1' :: Eff '[State String] ()
naide1' = get >>= \ s -> put ("Tere, " ++ s)
```

Joonis 2. Ühe efekti kasutamine freer-simple näitel

Mõlemad näited kasutavad sõne tüüpi seisundi monaadi ja lisavad sellele veel ühe sõne juurde. Kui on vaja kasutada mitmeid efekte funktsioonis, siis saab need komadega eraldada, nagu on toodud järgmises näites.

```haskell
naide2 :: Eff '[State String, Reader Int] ()
naide2 = do
  i <- ask
  put (replicate i '!')
```

Joonis 3. Mitme efekti kasutamine freer-simple näitel

Samas võib tekitada mitu erinevate tüüpidega seisundi monaadi, mis MTL teeki kasutades polnud lihtne teha [3]. Järgmisena tuleb need efektid töödeldada ja selleks on efektidel tavaliselt *run* nimega operatsioonid. Seisundi monaadil on näiteks *runState* meetod (`runState :: forall s effs a. s -> Eff (State s ': effs) a -> Eff effs (a, s)`), millel on `Eff effs (a, s)` tagastustüüp, kus *a* on tagastusväärtus ja *s* lõppväärtus.

```haskell
naide3 :: [a] -> Eff '[] ((), Int)
naide3 list = runState 0 $ do
  i <- get
  put (i + length list)
```

Joonis 4. Ühe efekti käivitamine freer-simple näitel

Eelnevas koodis on näha, et tüübi defineerimisel on järjend nüüd tühi ehk kõik efektid on läbitöödeldud. Kui on vaja mitu efekti käivitada, siis saab *run* meetodid jadamisi panna.

```haskell
naide4 ::  Eff '[] ((), Int)
naide4 = runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))
```

Joonis 5. Mitme efekti käivitamine freer-simple näitel

Freer-simple teegis on veel olemas *run* (`run :: Eff '[] a -> a`) ja *runM* (`runM :: Monad m => Eff '[m] a -> m a`) nimelised operatsioonid, mis tõstavad efekti monaadist välja ja teevad sellest puhta tagastustüübiga funktsiooni või tõstavad selle teise monaadi.

```
naide5 :: ((), Int)
naide5 = run . runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))

naide6 :: IO ((), Int)
naide6 = runM . runReader "tere" . runState 0 $ do
 list <- ask
 liftIO (putStrLn list)
 put (length list)
```

Joonis 6. Efekti monaadist väljatõstmine freer-simple näitel

Efektisüsteemidel on veel koodi paindlikkuse suhtes harjumus luua uusi efekte, mis töötaks programmi sees. Järgmises näites tehakse läbi uue *Teletype* efekti loomine, mis on üks tüüpiline näide efektisüsteemide selgitamisel.

```
data Teletype r where
  Read  ::           Teletype String
  Write :: String -> Teletype ()
-- Kasutades: {-# LANGUAGE TemplateHaskell #-}
makeEffect ''Teletype
```

Joonis 7. Teletype efekti loomine freer-simple näitel

Eelnevas koodis koostatakse uus *Teletype* tüüpi objekt, mis koosneb *Read* ja *Write* meetodite konstruktoritest. Kuid loodud *Teletype* objekti ei saa veel efektina kasutada, sest algul on vaja antud tüüp ühendada *Eff* efekti külge. Selleks on freer-simple teegiks olemas *makeEffect* meetod, mis automaatselt genereerib operatsioonid, kus iga *Teletype* meetoditele pannakse *send* (`send :: Member eff effs => eff a -> Eff effs a`) operatsioon külge. Alljärgnev on näide sellest, kuidas see sisemiselt välja näeb.

```
read'  :: Member Teletype r => Eff r String
read' = send Read
write' :: Member Teletype r => String -> Eff r ()
write' = send . Write
```

Joonis 8. Teletype meetodite käsitsi defineerimine freer-simple näitel

Järgnevalt on vaja koostada sellele efektile tema käitleja ja jällegi on selleks mitu valikut. Kui see efekt on juba mingis teises teegis olemas, siis saab kasutada *interpretM* (`interpretM :: forall eff m effs. (Monad m, LastMember m effs) => (eff ~> m) -> Eff (eff ': effs) ~> Eff effs`) meetodit ja tõlgendada see teise efekti jaoks.

```
runTeletype :: Eff '[Teletype, IO] a -> IO a
runTeletype = runM . interpretM $ \case
  Write msg -> Prelude.putStrLn msg
  Read -> Prelude.getLine
```

Joonis 9. Teletype meetodite loogika kirjutamine freer-simple näitel

Eelnevas koodis võetakse *Prelude* teegist *putStrLn* ja *getLine* operatsioonid ja ühendatakse need *Teletype* efekti külge.

Üheks kõige suuremaks miinuseks freer-simple teegil võrreldes teistega on selle kiirus [14]. MTL sisaldab küll $O(n^2)$ keerukust uute teisendajate koostamisel ja free-simple küll seda parandab, aga tänu sellele on see ka aeglasem tavalisest monaaditeisendajast ja ei sobi jõudlust nõudvatele rakendustele [12]. Üheks plussiks on välja toodud, et freer-simple teeb veateated mõne võrra paremaks, kuid algajatel võib nendest olla ikkagi raske aru saada.

3.2 fused-effects

Fused-effects on Haskelli tuntumaid efektisüsteeme, mille ülesandeks on sooritada efekte võimalikult tõhusalt. Fused-effects lisab võimaluse algebraliste ja kõrgema järgu efektide koostamiseks ning sisaldab üldkasutatavaid efekte [13].

Fused-effectsi paketid koosnevad peamiselt kahest osast: `Control.Effect`, mis määrab ära efekti tüübi ja selle võimalusi ning `Control.Carrier`, mis määrab ära kandja tüübi (ingl *carrier type*) ja käivitab antud efekte [13]. `Control.Effect` pakett koosneb sarnastest klassidest, mis on MTL teegis, kuid efekti tüüpe on rohkem ja funktsioonid on võimsamad. Efekti tegemiseks on fused-effectsis loodud *Has* nimega tüübipiirang [3]:

```
type Has eff sig m = (Members eff sig, Algebra sig m).
```

Has nimelise tüübi signatuuris tähistab *eff* mingit kindlat efekti, *sig* efektide järjendid ja *m* kindlat monaadi. Muutuja *m* on nagu konteiner, mis hoiab endas *sig* järjendit. Programmi käivitamisel peaksid kõik need efektid olema ära sooritatud.

Nagu eelnevalt öeldi, siis järgmised näited on kasutuses fused-effectsi dokumentatsioonis [13] ning need on sarnased eelmiste näidetega, aga kasutades fused-effectsi süntaksit.

```
naide1 :: Has (State String) sig m => m ()
naide1 = get >>= \ s -> put ("Tere, " ++ s)
```

Joonis 10. Ühe efekti kasutamine fused-effects näitel

Eelnevas näites `Has (State String) sig m` nõuab, et *m* monaad koosneks seisundi monaadist. Sellise koodi näite kompileerimiseks on vaja algusesse veel kirjutada

`import Control.Carrier.State.Strict`, mis impordib vajalikud *State* meetodid ja `Control.Effect` paketi ise pole vaja importida.

Kui on vaja mitu efekti sooritada, siis tuleb *Has* tüüpi mitu korda järjest panna.

```
naide2 :: (Has (State String) sig m,
           Has (Reader Int) sig m) => m ()
naide2 = do
  i <- ask
  put (replicate i '!')
```

Joonis 11. Mitme efekti kasutamine fused-effects näitel

Naide2 meetodil esineb mitu *Has* tüübi piirangut, mis on omavahel komadega eraldatud ja need nõuavad *State* ja *Reader* monaadide olemasolu.

Kui on vaja tehtud efektid käivitada, siis on `Control.Carrier` pakettides olemas *run* nime algusega meetodid, mis käivitavad vastavaid efekte.

```
naide3 :: Algebra sig m => [a] -> m (Int, ())
naide3 list = runState 0 $ do
  i <- get
  put (i + length list)
```

Joonis 12. Ühe efekti käivitamine fused-effects näitel

```haskell
naide4 :: Algebra sig m => m (Int, ())
naide4 = runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))
```

Joonis 13. Mitme efekti käivitamine fused-effects näitel

Naide3 ja *naide4* funktsioonides käivitatakse välja toodud efektid ja *naide4* pealt on näha, kuidas saab toimida mitme efekti korral. Nendes olukordades tuleb meetodile lisada `Algebra sig m` tüübipiirangu.

Kui kõik meetodid on juba käivitatud, siis on võimalik neid tõsta fused-effectsi monaadist välja ja paigutada neid puhta funktsiooni või teise monaadi sisse. Selle jaoks on fused-effectsi teegis samamoodi olemas *run* ja *runM* meetodid ning nende kasutus on sarnane freer-simple omaga.

```haskell
naide5 :: (Int, ())
naide5 = run . runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))

naide6 :: IO (Int, ())
naide6 = runM . runReader "tere" . runState 0 $ do
  list <- ask
  liftIO (putStrLn list)
  put (length list)
```

Joonis 14. Efekti monaadist väljatõstmine fused-effects näitel

Järgmises näites tuuakse välja, kuidas saab *Teletype* efekti luua fused-effects teegiga.

```haskell
data Teletype (m :: Type -> Type) k where
  Read  ::           Teletype m String
  Write :: String -> Teletype m ()

read :: Has Teletype sig m => m String
read = send Read
write :: Has Teletype sig m => String -> m ()
write s = send (Write s)
```

Joonis 15. Teletype efekti loomine fused-effects näitel

Algul luuakse uus *Teletype* tüüp, kus on *Read* ja *Write* nimedega konstruktorid. Nende kaudu luuakse meetodid, kus *send* meetod tõstab need fused-effectsi koostatud monaadi. Sellega on uus efekti definitsioon loodud ja sellele tuleb lisada efektikäitleja (ingl *effect handler*).

```
newtype TeletypeIOC m a = TeletypeIOC { runTeletypeIO :: m a }
  deriving (Applicative, Functor, Monad, MonadIO)

instance (MonadIO m, Algebra sig m) =>
  Algebra (Teletype :+: sig) (TeletypeIOC m) where
  alg hdl sig ctx = case sig of
    L Read      -> (<$ ctx) <$> liftIO getLine
    L (Write s) -> ctx      <$  liftIO (putStrLn s)
    R other     -> TeletypeIOC (alg (runTeletypeIO . hdl) other ctx)
```

Joonis 16. Teletype meetodite loogika kirjutamine fused-effects näitel

Selle ülesande jaoks on eelnev kood, mis samuti kasutab *getLine* ja *putStrLn* funktsioone ning viib nende efektid üle *Teletype* objektile.

Võrreldes teiste tuntud efektisüsteemidega on fused-effects teek kiirem ja väga sarnane MTL enda kiirusega [14]. Selle tõttu on fused-effectsi hea kasutada sellistes olukordades, kus programmi töökiirus on oluline, aga MTL võimekusest päris ei piisa. Fused-effects on loodud spetsiaalselt väga sarnaselt MTL teegiga ja koodi ümberkirjutamine ühe teegi pealt teisele nõuab minimaalseid muudatusi [13]. Fused-effectsi teegil puudub $O(n^2)$ probleem ning võimaldab funktsioonis mitme sama tüübi monaadi defineerimist. Võrreldes freer-simple ja polysemy teegiga on fused-effects väga sarnane, kuid see töötab kiiremini ja ei pane eriti rõhku veateadete optimeerimisele.

3.3 polysemy

Polysemy on veel üks populaarne efektisüsteemi põhine Haskelli teek. Nagu mõned teisedki efektisüsteemid on seegi Oleg Kiselyov ja Nicolas Wu teadustööde põhjal koostatud või nendest idee saanud [16]. Polysemy lubab kirjutada võimsat ja tõhusat efekti põhilist koodi ning võimaldab eraldada efekti loogikat ja rakendust, mille tõttu saavutatakse väheste kordustega kood. Polysemy teegis on erilist tähelepanu pööratud kasutussõbralikusele ja veasõnumid on tehtud palju lihtsamaks ning arusaadavamaks, mis on kohati Haskelli süsteemis päris suureks probleemiks.

Polysemy koodi süntaks on väga sarnane freer-simple omaga, kuid meetodite nimetused võivad kohati erineda [11, 15]. Järgmistes näidetes võib näha, et polysemy teegis kasutatakse efektide

hoidmiseks `Sem effs r` monaadi ja efektide lisamist saab paremini sooritada *DataKinds* laiendi kasutusega.

```
naide1 :: Member (State String) effs => Sem effs ()
naide1 = get >>= \ s -> put ("Tere, " ++ s)

-- Kasutades: {-# LANGUAGE DataKinds #-}
naide1' :: Sem '[State String] ()
naide1' = get >>= \ s -> put ("Tere, " ++ s)

naide2 :: Sem '[State String, Reader Int] ()
naide2 = do
  i <- ask
  put (replicate i '!')
```

Joonis 17. Efektide kasutamine polysemy näitel

Efektide käivitamine toimub jällegi sarnaselt freer-simple teegiga, kuid väheseid erinevusi ikkagi leidub. Kui vaadata *runState* meetodi definitsiooni (`runState :: s -> Sem (State s : r) a -> Sem r (s, a)`) ja võrrelda seda freer-simple omaga, siis tagastusväärtuse järjekord neil erineb.

```
naide3 :: [a] -> Sem '[] (Int, ())
naide3 list = runState 0 $ do
  i <- get
  put (i + length list)

naide4 ::  Sem '[] (Int, ())
naide4 = runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))
```

Joonis 18. Efektide käivitamine polysemy näitel

Naide5 ja *naide6* meetodid, mis eelmistes peatükkides defineeriti, on samasuguse kujuga polysemy teegis.

Sarnaselt eelmiste teekidega tuleb uue efekti loomiseks algul defineerida uus efekti objekt vajaliku tüübi ja konstruktoritega. Kuid praeguseks seda polysemy teegiga kasutada ei saa ja selleks on olemas *makeSem* (`makeSem :: Name -> Q [Dec]`) meetod, mis automaatselt loob selle jaoks vastavad meetodid *Sem r a* tagastustüübiga.

```haskell
data Teletype m k where
  Read  ::           Teletype m String
  Write :: String -> Teletype m ()
-- Kasutades: {-# LANGUAGE TemplateHaskell #-}
makeSem ''Teletype
```

Joonis 19. Teletype efekti loomine polysemy näitel

Järgmisena on vaja kirjutada *Teletype* efektikäitleja ja selleks saab kasutada *interpret* ja *embed* (`embed :: Member (Embed m) r => m a -> Sem r a`) funktsioone, mis tõstavad efetki *Sem* tüüpi.

```haskell
teletypeToIO :: Member (Embed IO) r => Sem (Teletype ': r) a -> Sem
r a
teletypeToIO = interpret \case
  Read      -> embed getLine
  Write msg -> embed $ putStrLn msg
```

Joonis 20. Teletype meetodite loogika kirjutamine polysemy näitel

Polysemy GitHubi [15] lehel on veel toodud näitena, kuidas selle efektile käivitusfunktsiooni saaks kirjutada.

```haskell
runTeletypePure :: [String] -> Sem (Teletype ': r) a
                    -> Sem r ([String], a)
runTeletypePure i =
      runOutputMonoid pure . runInputList i . reinterpret2 \case
      Read -> maybe "" id <$> input
      Write msg -> output msg
```

Joonis 21. Teletype käivitusfunktsiooni kirjutamine polysemy näitel

Eelnev näide kasutab `Polysemy.Input` ja `Polysemy.Output` paketis olevaid funktsioone, et hoida sõne järjendis *Telepyte* efekti sisendeid ja väljundeid.

Kui võrrelda teiste teekidega, siis MTL teegiga on polysemy efektiivsem, nõuab vähem kordusi koodis ja selles puudub $O(n^2)$ isendi probleem, mida üleval juba kirjeldati [15]. Võrreldes freer-simple teegiga on polysemy rohkemate võimalustega ja paindlikum. Fused-effects on sisemiselt sellega sarnane, kuid polysemy kasutades saab koodi efektiivsemalt kirjutada.

3.4 effet

Effet on üks uuemaid efektisüsteeme ja sarnaselt eelmise teegiga tuuakse välja, et sellel on võimalik eraldi kirjutada efekti definitsioon ja efektikäitleja [16]. Effet teeki on proovitud koostada võimalikult lihtsalt, et pärast oleks koodist kergem aru saad. Sisemiselt effet kasutab peamiselt olemasolevat *transformers* teeki koos mõnede lisadega ja sarnaneb väga palju teiste efektisüsteemide teekidega, näiteks sellel on sarnane tüübiklasside süsteem, mis on MTL ja fused-effectsis ja sarnane efektide loomise meetod, mis on polysemy teegis [16].

Efekte kasutatavate meetodite defineerimisel saab rakendada `Monad m => m ()` struktuuriga tüübipiirangut, kus *m* on effet teegi monaad. Eelmiste peatükkide järgi demonstreerivad järgmised näited, kuidas on võimalik efekte kasutada funktsioonides.

```
naide1 :: State String m => m ()
naide1 = get >>= \ s -> put ("Tere, " ++ s)

naide2 :: (State String m, Reader Int m) => m ()
naide2 = do
  i <- ask
  put (replicate i '!')
```

Joonis 22. Efektide kasutamine effet näitel

Sarnaselt teiste teekidega on võimalik monaade käivitada kasutades *run* funktsioone, mida demonstreerib järgmine näide.

```
naide3 :: State String m => [a] -> m (Int, ())
naide3 list = runState 0 $ do
  i <- get
  put (i + length list)

naide4 :: (State String m, Reader Int m) => m (Int, ())
naide4 = runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))
```

Joonis 23. Efektide käivitamine effet näitel

Kui kõik efektid on ära käivitatud ja järele on jäänud `Monad m => m a` tüübi kuju, siis sellest on võimalik monaad välja tõsta ja järele jääb puhas funktsioon. Selle jaoks on effet teegis

olemas *Identity* monaad, mis sisaldab *runIdentity* (`runIdentity :: Identity a -> a`) meetodit.

```
naide5 :: (Int, ())
naide5 = runIdentity . runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))
```

Joonis 24. Efekti monaadist väljatõstmine effet näitel

Järgmistes näidetes luuakse uuesti *Teletype* efekti ja kirjutatakse selle käivitusfunktsioon.

```
class Monad m => Teletype' tag m where
  read'  ::           m String
  write' :: String -> m ()
makeTaggedEffect ''Teletype'
```

Joonis 25. Teletype efekti loomine effet näitel

Effet teek lubab veel tavalise efekti asemel luua märgistatud efekte (ingl *tagged effect*), mille kaudu saab mitmeid sama nimelisi efekte kergesti eristada [16]. Efekti loomisel nõuab see *tag* muutuja lisamist, mida saab pärast efekti loomisel väärtustada. Selline efekt luuakse eelmises näites. Sarnaselt polysemy'ga on võimalik effet teegis kasutada *makeTaggedEffect* operatsiooni, mis koostab automaatselt vajalikud meetodid efekti loomiseks.

Järgmisena tuleb koostada *Teletype* efektikäitleja, et pärast saaks neid käivitada.

```
newtype TeletypeIOC m a = TeletypeIOC { runTeletypeIO :: m a }
  deriving (Applicative, Functor, Monad, MonadIO)
instance MonadIO m => Teletype' tag (TeletypeIOC m) where
  read'    = liftIO $ getLine
  write' s = liftIO $ putStrLn s
```

Joonis 26. Teletype meetodite loogika kirjutamine effet näitel

Kasutades *liftIO* (`liftIO :: IO a -> m a`) meetodit tõstetakse olemasolevad efektid nõutud funktsioonidesse. Järgmisena on loodud vajalikud käivitus meetodid, kus üks on märgistatud ja teine tavalise versiooni jaoks.

```haskell
-- import Data.Coerce (coerce)
runTeletype' :: (Teletype' tag `Via` TeletypeIOC) m a -> m a
runTeletype' = coerce

makeUntagged ['runTeletype']
```

Joonis 27. Teletype käivitusfunktsiooni kirjutamine effet näitel

Nagu eelnevalt öeldi, siis effeti teek on välimuselt mitme efektisüsteemi kombinatsioon. Piiranguteks on teegi autor välja toonud, et kõikides olukordades *TemplateHaskell* laienduse abifunktsioonid, näiteks *makeEffect* ja *makeTaggedEffect*, ei pruugi töötada ja peab neid ise defineerima [16]. Teegi kiirust pole ära mõõdetud, aga teegi autor arvab, et see on kuskil MTL sarnaste numbritega.

4. Efektisüsteemi õppematerjalid

Eelmistes peatükkides kirjeldati nelja erinevat efektisüsteemi, mis Haskellis on tehtud, toodi välja nende head ja halvad küljed ning võrreldi neid omavahel. Nende tulemuste põhjal on autor valinud fused-effectsi teegi ja koostanud selle põhjal efektisüsteemide õppematerjalid. Fused-effects on praegusel hetkel üks kõige enim kasutatud efektisüsteem Haskelli keskkonnas ja selle tulemusel on fused-effectsi ümber käiv kommuun suhteliselt suur. Teeki uuendatakse regulaarselt ja olemasolev dokumentatsioon on autori arvates parem kui paljudel teistel teekidel. Fused-effects on väga paindlik ja kiiruse poole pealt sarnane MTL teegiga. See teek suudab lahendada enamus asju, mis teised efektisüsteemi teegid ja tudengil võib sellest kasu olla edasistes õpingutes. Efektisüsteemi kasutus arendab arusaama Haskelli keskkonnast ja sellega on võimalik programme efektiivsemalt kirjutada. Valminud õppematerjalid on lisatud selle töö lisadesse.

4.1 Õppematerjali kontekst

“Programmeerimiskeelte” aines toimub kokku 16 loengut ja 16 praktikumi. Kust ainult pooled käsitlevad Haskelli teemalisi materjale. Loengutes tutvustatakse antud teemasid ja praktikumides tehakse nende peal ülesandeid. Praktikumide jaoks on koostatud materjalid, mis alguses sisaldavad näiteülesandeid, millele eelneb selle näite väike tutvustus. Pärast tulevad harjutusülesanded, kus algul tudeng proovib need ise lahendada ja pärast lahendatakse need ära koos praktikumi juhendajaga. Praktikumi juhendaja ülesandeks on seletada, kuidas Haskelli süsteemis saab programmeerida ja aidata tudengitel lahendada ülesanded. Mõned praktikumi materjalid sisaldavad veel raskemaid ülesandeid, mis on tavaliselt jäetud iseseisvaks lahendamiseks. Lisaks on mõnede praktikumide jaoks loodud kodutööd, kus tuleb rakendada praktikumis saadud teadmisi.

Antud õppematerjalid on mõeldud üheks viimaseks teemaks Haskellit katvas materjalis ja nendega võib korraldada viimase Haskelli praktikumi või anda tudengitele lisamaterjalina. Selleks ajaks on tudengid juba tutvunud tüübiklassidega ja oskavad kasutada monaade funktsioonide sees. “Programmeerimiskeelte” aine viimases loengus küll mainitakse monaaditeisendajat, aga ei minda selle teemaga sügavamale. Kuid monaadide kasutamise teadmistega on võimalik antud õppematerjalid juba läbida. Kui õppematerjale kasutatakse praktikumi läbiviimiseks, siis juhendajal on võimalik kõik näited ära seletada ning näidata erinevaid lahendusviise ülesannetele.

4.2 Õppematerjalide kirjeldus

Õppematerjalide koostamisel võeti arvesse, et kõik materjalide lugejad võib-olla ei mäleta, mis puhtad funktsioonid ja kõrvalefektid olid ning koostati selle kohta lühitutvustus. Sealses tekstis toodi välja monaadi kasutus ja räägiti mõnest populaarsemast monaadist. Edasised lõigud kirjeldavad õppematerjalides vanemaid lahendusi monaadide kokku panemiseks, milleks on MTL teegi kasutus, ja kuidas saaks teha veel paremini.

Õppematerjali esimesed lõigud andsid ülevaate varasemast monaadi teooriast ja järgmistes lõikudes räägitakse puhtalt fused-effectsi teegist. Selleks toodi välja fused-effectsi lühikirjeldus ja selle kasulikkus oma programmide koostamises. Järgmiseks teemaks õpetati tudengil fused-effectsi kasutamist oma projekti keskkonnas, milleks loodi projekt kasutades hpack [17] laiendust. Hpack on Haskelli laiendus, mille kaudu on võimalik lisada oma projekti erinevaid Haskelli teeke. Ennem kui minnakse efekti kasutamise näidete juurde räägitakse fused-effectsi sisemisest struktuurist, et oleks kergem aru saada, millised paketid tuleks importida oma projekti efektide kasutamiseks.

Edasiselt räägitakse efekte kasutatavate meetodite sisse toomisest ja nende käivitamisest. Nende meetodite demonstreerimiseks on kasutatud samu näited, mis olid selles töös ja vahepeal on seletatud, kuidas need efektid töötavad. Sarnaselt eelmistele “Programmeerimisekeelte” aine materjalidele on näidete järel olemas efektidest koosnevad ülesanded. Alguse ülesanded on lihtsamad, et õppematerjalide lugeja harjuks ära fused-effectsi süntaksiga. Nende ülesannete lahendamiseks on mitu võimalust ja praktikumi juhendaja peaks välja tooma need erinevad viisid, et tekiks sügavam arusaam fused-effectsi teegiga ja üldise efektisüsteemide struktuuriga.

Näitena on kasutusele võetud fused-effects-mwc-random [18] teegist Random monaad, mis laseb luua pseudojuhuslike numbreid. Sellise meetodiga näidatakse, kuidas saab kasutada teisi fused-effectsi laiendusi ja saab proovida mitte-standardsete monaadide kasutust. Selle laienduse demonstreerimiseks on selle kohta kirjutatud tutvustus ja loodud lühikesed ülesanded, kus peab looma järjendi juhuslikest numbritest.

Kõige viimase teemana käsitletakse uute efektide loomist ja tuuakse näitena *Teletype* efekti loomise. See on kõige keerulisem osa õppematerjalides ja koodi struktuurist aru saamise juures oleks abiks praktikumi juhendaja seletused, kuid funktsioonide tüüpide uurimise käigus on võimalik sellest ka iseseisvalt aru saada. Samalaadse struktuuri järgi on ülesandeks teha *Printer* efekt, mis suudab teksti välja trükkida ja teksti kirjutada etteantud faili. Selle jaoks soovitakse koostada efektikäitlejad *printFile* ja *printConsole* meetoditele.

Viimaseks ülesandeks on autor loonud sellise ülesande, mis võimaldaks kasutada kõiki eelnevaid ülesannete lahendusi ja selle kaudu moodustada meetod, mis tekitab etteantud number juhuslikke arve ning kirjutab need mingisse faili. Sellisel tulemusel on võimalik näha, kuidas saab luua oma efekti ja kasutada need koos teiste efektidega, et moodustada mingi funktsioon. Nende ülesannete baasil on võimalik pärast koostada kodutöö, kus kontrollitakse tudengi saadud teadmisi.

4.3 Ülesannete valik

Õppematerjalid on koostatud selliselt, et põhilised fused-effectsi funktsioonid oleksid läbi vaadatud. Selleks on kirjeldatud efekti kasutust, selle käivitamist ja uue efekti loomist. Ülesanded on võetud sellised, et need toetaksid nendele teemadele ja aitaksid tudengil selgeks saada nende süntaksi kirjutamist. Alguses on toodud lihtsamad näited seisundi monaadi kasutamisest efektisüsteemis ja pärast tuuakse sisse Random monaad, et näidata teise kasuliku monaadi kasutamist. Sellega võiks juba tudengil tekkida arusaam, milleks on efektisüsteemid kasulikud ja kuidas saaks neid teistes kohtades kasutada.

Uue efekti loomise näide on võetud selliselt, et need oleksid pärast kasulikud teiste samalaadsete projektide jaoks. Selleks on mingi faili kirjutamine ja teksti väljatrükk konsooli. Selliseid efekte on sagedasti vaja suuremates programmides ja nende kasutuse kohta võib leida igasuguseid näited.

Viimane ülesanne on valitud selliselt, et see kinnitaks õppematerjalides olevad teemad. Selleks tuleb luua selline meetod, kus tehakse mitme juhusliku numbri loomist, selle välja trükkimist ja faili kirjutamist. Selle lahenduseks saaks kasutada eelmistes näidetes loodud efektid ning panna need kokku sobiva kujuga.

Ülesannete läbimisel peaks tudeng oskama koostada lihtsaid efekte ja neid kasutada oma programmides. Samas annab õppematerjal teadmisi efektisüsteemide kohta ja võimaldab tudengil kergemini aru saada teistest teekidest, mis samuti kasutavad sarnast metoodikat.

Kokkuvõte

Töö eesmärgiks oli koostada õppematerjalid efektisüsteemide õpetamiseks Haskellis. Selle töö käigus vaadeldi nelja erinevat efektisüsteemi (freer-simple, fused-effects, polysemy ja effet) ja nende seast valiti üks, mille kohta moodustati õppematerjalid.

Töö jagati suuresti kolmeks osaks. Esimeses osas räägiti ülesande püstitusest. Seletati ära funktsionaalse programmeerimise omapärad ja tutvustati Haskelli keelt ning monaadide esitust. Järgmises toodi välja neli erinevat efektisüsteemi teeki ja toodi näited nende kasutamise kohta. Viimases osas vaadati õppematerjalide koostamist ja räägiti ülesannete valikust.

Selle töö käigus valmis õppematerjal, kus õpetati fused-effectsi teegi baasil efektisüsteeme. Sellega seoses räägiti efektide kasutamisest funktsioonides ja õpetati neid käivitama programmides. Veel näidati uute efektide loomist, et tekiks arusaam efektisüsteemide kasulikkusest. Õppematerjali näidete juurde koostati ülesanded, mille läbimisel tudeng peaks oskama kirjutada lihtsaid efekte kasutatavaid meetodeid ja rakendada neid oma projektides.

Valminud õppematerjale saab näiteks kasutada “Programmeerimiskeelte” kursusel Haskelli õpetamiseks või ka teistes kursustes lisamaterjalina. Lisaks aitas see autoril paremini Haskelli keelest aru saada ning täiendas oskusi teiste samalaadsete süsteemide koostamise jaoks.

Viidatud kirjandus

[1] Ainekava info kursusel “Programmeerimiskeeled”. https://www.is.ut.ee/rwservlet?oa_ainekava_info.rdf+1367120+PDF+0+application/pdf (05.05.2021)

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

[3] Diehl S. What I Wish I Knew When Learning Haskell. http://dev.stephendiehl.com/hask/tutorial.pdf (05.05.2021)

[4] Jones M.P. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*, 1995, p. 97-136. https://web.cecs.pdx.edu/~mpj/pubs/springschool95.pdf (05.05.2021)

[5] Schrijvers T., Piróg M., Wu N., Jaskeliof M. Monad transformers and modular algebraic effects: what binds them together. *12th ACM SIGPLAN International Symposium on Haskell*, 2019, p. 98-113. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW699.pdf (05.05.2021)

[6] MTL teegi lähtekood. https://github.com/haskell/mtl (05.05.2021)

[7] MTL teegi Class.hs lähtekood. https://github.com/haskell/mtl/blob/master/Control/Monad/Reader/Class.hs (05.05.2021)

[8] Orchard D., Petricek T. Embedding effect systems in Haskell. 2014. https://www.doc.ic.ac.uk/~dorchard/publ/haskell14-effects.pdf (05.05.2021)

[9] Figueroa I., Leger P., Fukuda H. Which monads Haskell developers use: An exploratory study. *Science of Computer Programming*, 2020. https://www.sciencedirect.com/science/article/pii/S0167642320301313 (05.05.2021)

[10] Kiselyov O., Sabry A,. Swords C., Extensible effects: an alternative to monad transformers. 2013. http://okmij.org/ftp/Haskell/extensible/exteff.pdf (05.05.2021)

[11] Freer-simple teegi lähtekood. https://github.com/lexi-lambda/freer-simple (05.05.2021)

[12] Maguire S. Freer Monads, More Better Programs. 2019. https://reasonablypolymorphic.com/blog/freer-monads/ (05.05.2021)

[13] Fused-effects teegi lähtekood. https://github.com/fused-effects/fused-effects (05.05.2021)

[14] Kiiruste testimine erinevate Haskelli efektisüsteemidega. https://github.com/patrickt/effects-benchmarks (05.05.2021)

[15] Polysemy teegi lähtekood. https://github.com/polysemy-research/polysemy (05.05.2021)

[16] Effet teegi lähtekood. https://github.com/typedbyte/effet (05.05.2021)

[17] Fused-effects-mwc-random laienduse dokumentatsioon. https://hackage.haskell.org/package/fused-effects-mwc-random (05.05.2021)

Lisad

I. Õppematerjalid

Nagu juba kursuse algusest peaks teadma, siis Haskell on puhas keel. Kõik funktsioonid, mis seal saab defineerida peavad olema puhtad ehk mingeid kõrvalefekte ei tohiks toimuda ja Haskell seda ka väga hästi garanteerib. Sellised funktsioonid on väga piiratud, aga need on omakorda väga turvalised ja annavad iga sisendi korral kindla vastuse. Kui puhta funktsiooni käivitad, siis see võib kas tagastada mingi kindla väärtuse või võib jääda lõpmatult käima. Aga sellised puhtad matemaatilised funktsioonid on suhteliselt igavad ja mingit interaktiivset asja seal on väga raske koostada. Selle jaoks on Haskelli arendajad välja mõelnud IO monaadi, mille sees on juba kõige võimalik teha. IO monaadis saab sooritada teksti väljatrükki, kõvaketta pealt faile lugeda, internetiga ühenduda jne. Kuid tuleks ikkagi tähele panna, et Haskellit loetakse puhtaks keeleks ning kõrvalefektid on väga kavalalt rakendatud, et need töötaksid ilusti selles keeles. Haskelli süsteemis on peale IO monaadi veel teisigi monaade, näiteks Reader monaad on keskkonna ehk konfiguratsiooni lugemise monaad ja State monaadiga saab keskkonnast väärtusi lugeda ja kirjutada. Parema töö saavutamiseks peaks mitu monaadi kokku panema.

Üheks lahenduseks sellele probleemile on olnud monaaditeisendajad. Kõige populaarsem teek, mis neid lisab on MTL. Kuid sellel on jälle mõningad probleemid, mis võivad natukene raskendada programmeerimist. Selle alternatiiviks on loodud efektisüsteemid. Need on tavaliselt paindlikumad, kui monaaditeisendajad ja elimineerivad mõningaid nende probleeme.

Fused-effects on üks kõige populaarsemaid teeke, mis kasutab efektisüsteemide tehnoloogiat monaadide kokkupanemiseks ja see on tugev tööriist programmide koostamisel. Fused-effectsi GitHubi lehel on väga hästi kirjeldatud erinevad funktsioonid efektidega ja probleemide korral tungivalt soovitatav see läbi vaadata.

Alguses on vaja luua uus projekti keskkond, kus saaks fused-effectsi kasutada.

```
stack new --resolver lts-17.10 fused-program simple-hpack
```

Eelnev käsklus loob projekti, kus on olemas veel lihtsustatud hpack laiendus. Selle tulemusel tekitatakse **package.yaml** fail ja sinna tuleb lisada:

```
extra-deps:
  - git: https://github.com/fused-effects/fused-effects.git
    commit: be44e4ffbca238acdc4c7cccececb3b656db81f6e
  - git: https://github.com/fused-effects/fused-effects-mwc-
random.git
    commit: d0d765e2842e980b9029a574693d99a622867ebb
```

Peale selle tuleb muuta **stack.yaml** failis read:

```
default-extensions:
  - FlexibleInstances
  - GeneralizedNewtypeDeriving
  - MultiParamTypeClasses
  - TypeOperators
  - UndecidableInstances

dependencies:
  - base >= 4.7 && < 5
  - fused-effects
  - fused-effects-mwc-random
```

Selle tulemusel luuakse projekt, kus on võimalik kasutada fused-effectsi ja selle laiendusi.

Fused-effects on jaotatud kolme paketi vahele:

- *Control.Effect*, mis sisaldab endas efekte ja funktsioone, mille kaudu saab neid defineerida.
- *Control.Carrier*, mis sisaldab funktsioone antud efektide käivitamiseks.
- *Control.Algebra*, mis sisaldab vajalikku algebrat uute efektide loomiseks.

Mingi monaadi lisamiseks tavaliselt piisab ainult *Control.Carrieri* paketis olevast meetodi importimisest.

Efektide sissetoomine

Efektid hoitakse *m* nimelises konteineris ja nende sisalduse kontrolliks tuleb fused-effectsis *Has* tüüpiga piirangut sooritada.

```
naide1 :: Has (State String) sig m => m ()
naide1 = get >>= \ s -> put ("Tere, " ++ s)

naide2 :: (Has (State String) sig m,
           Has (Reader Int) sig m) => m ()
naide2 = do
  i <- ask
  put (replicate i '!')
```

Efektide kävitamine

Kui loodud efekte on vaja käivitada, siis selleks on igal monaadil erinevad *run* operatsioonid olemas. Näiteks *State* monaadil on *runState*, *evalState* ja *execState*, mis sooritavad antud efekti, aga tagastavad erinevaid väärtusi. Mitme efekti korral saab neid funktsioone järjest lisada ja pärast need ükshaaval käivitada.

```
naide3 :: Algebra sig m => [a] -> m (Int, ())
naide3 list = runState 0 $ do
  i <- get
  put (i + length list)

naide4 :: Algebra sig m => m (Int, ())
naide4 = runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))
```

Kui kõik efektid on käivitatud, siis võib tekkida tahtmist need monaadi tüübist välja tõsta või mingisse teise monaadi paigutada ja selleks on *run* ning *runM* operatsioonid.

```haskell
naide5 :: (Int, ())
naide5 = run . runReader "tere" . runState 0 $ do
  list <- ask
  put (length (list :: String))

naide6 :: IO (Int, ())
naide6 = runM . runReader "tere" . runState 0 $ do
  list <- ask
  liftIO (putStrLn list)
  put (length list)
```

Järgmiseks võiks proovida luua efekti, mis võtaks State monaadist numbrite järjendi, leiaks järjendi elementide summa ja see number kirjutatakse State monaadi järjendi lõppu.

```haskell
ex1 :: Has (State [Int]) sig m => m ()
ex1 = undefined

exRun1 :: Algebra sig m => [Int] -> m [Int]
exRun1 list = undefined

exRun2 :: [Int] -> [Int]
exRun2 list = undefined
```

Veel üheks kasulikuks efektiks on juhuslike numbrite loomine. Selle jaoks on projektis lisatud veel [fused-effects-mwc-random] teek, mis lubab igasuguste meetodite järgi pseudojuhuslike numbreid genereerida. Selle jaoks on paketis näiteks olemas *uniformR* ja *runRandomSystem* meetodid, mille definitsioone on soovituslik vaadata teegi dokumentatsioonist. Järgmiste ülesannete käigus tuleb teha efekt, mis genereerib juhuslikult numbreid (vahemikus 1-100) ja *ex4* ülesandes peaks moodustuma järjend, milles on *n* juhuslikku arvu.

```haskell
ex2 :: Has Random sig m => m Int
ex2 = undefined

ex3 :: IO Int
ex3 = undefined

-- Soovitav kasutada replicateM meetodit
-- Defineeritud: 'Control.Monad' paketis
ex4 :: Int -> IO [Int]
ex4 n = undefined
```

Efektide loomine

Kõige tähtsamaks osaks on uute efekte loomine ning selleks on vaja algul luua uus efekti tüüp, kus oleks vajalike meetodite konstruktorid. Iga meetodi defineerimisel tuleb kasutada *send* funktsiooni.

```
data Teletype (m :: Type -> Type) k where
  Read  ::           Teletype m String
  Write :: String -> Teletype m ()

read :: Has Teletype sig m => m String
read = send Read

write :: Has Teletype sig m => String -> m ()
write s = send (Write s)
```

Kui meetodid on defineeritud tuleb nendele kirjutada vastav algebra ja selle jaoks saab kasutada järgnevat kuju.

```
newtype TeletypeIOC m a = TeletypeIOC { runTeletypeIO :: m a }
  deriving (Applicative, Functor, Monad, MonadIO)

instance (MonadIO m, Algebra sig m) =>
          Algebra (Teletype :+: sig) (TeletypeIOC m) where
  alg hdl sig ctx = case sig of
    L Read      -> (<$ ctx) <$> liftIO getLine
    L (Write s) -> ctx      <$  liftIO (putStrLn s)
    R other     -> TeletypeIOC (alg (runTeletypeIO . hdl) other ctx)
```

Järgmisena võiks ise proovida koostada Printer efekt, mille kaudu saaks sooritada väljatrüki konsooli aknasse ja kirjutada mingisse faili. Aluskood on juba ette antud.

Faili kirjutamise jaoks on saadaval *System.IO* paketis olev meetod *writeFile*, mida selles lahenduses saab kasutada.

```
data Printer (m :: Type -> Type) k where
  PrintFile    :: FilePath -> String -> Printer m ()
  PrintConsole ::             String -> Printer m ()

printFile :: Has Printer sig m => FilePath -> String -> m ()
printFile path text = undefined
...
```

Efekti valmis saamisel peaks olema võimalik defineerida meetod, mis koos eelneva juhusliku numbri generaatoriga suudab mingisse kindlasse faili kirjutada *n* suvalist numbrit ja samal ajal need ekraanile väljastada.

```
printerEx :: Int -> String -> IO ()
printerEx n path = undefined
```

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Peter Kallaste**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   Efektisüsteemide õpetamine Haskellis,

   mille juhendaja on **Kalmer Apinis**,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Peter Kallaste

07.05.2021