TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Marti Mutso

Haxl teegi uurimine

Bakalaureusetöö (9 EAP)

Juhendaja: Kalmer Apinis

Tartu 2017

Haxl teegi uurimine

Lühikokkuvõte:
Selles lõputöös tutvustatakse ning demonstreeritakse päringute optimeerimise teeki *Haxl*. Teegi kasutamine, põhilised funktsionaalsused ning nende töötamise loogika tuuakse välja, luues kaks näidisprogrammi.

Lisaks antakse *Haxl'*i kohta üldisem ülevaade, seejuures olulisemad *Haskell'*i osad, mida teegi loojad *Haxl'*i jaoks kasutasid. Lühidalt antakse ülevaade ka *Haskell Database Connectivity* teegi kohta.

Töö tulemusena valmib kaks *Haxl* teeki tutvustavat näidisprogrammi ja nende seletused, mis demonstreerivad teegi kasutamist ning mille abil on võimalik lugejal otsustada teegi kasulikkuse ning keerukuse üle.

Võtmesõnad: funktsionaalne programmeerimine, Haskell, Haxl, päringute optimeerimine

CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, kontroll

Researching the Haxl library

Abstract:
This Bachelor's thesis demonstrates and gives an overview of *Haxl*, a Haskell library for optimizing queries. The different features of the library will be explained by creating two sample programs.

In addition, a more general overview will be given, such as why the library was created in the first place. *Haskell Database Connectivity* library will also be briefly covered.

The result of this thesis is the two created sample programs, which with their explanations will allow the reader to decide on the usefulness and practicality of the library.

Keywords: functional programming, Haskell, Haxl, query optimizing

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

Sisukord

1. Sissejuhatus....4
2. *Haxl* teek....6
3. Algeline näidisprogramm....7
   3.1. Idee....7
   3.2. Päringute jooksutamine....7
   3.3. Andmeallika loomine....9
4. Keerulisem näidisprogramm....12
   4.1. Idee....12
   4.2. *Haskell Database Connectivity* teek....12
   4.3. Päringute jooksutamine....13
   4.4. Andmeallikate loomine....16
5. Kokkuvõte....20
6. Viidatud kirjandus....21
7. Lisad....23
   7.1. Näidisprogramm 1....23
      7.1.1. *NaideMain.hs*....23
      7.1.2. NaideDataSource.hs....23
   7.2. Näidisprogramm 2....24
      7.2.1. *StoreMain.hs*....24
      7.2.2. *StoreHaxl.hs*....25
      7.2.3. *StoreDataSource.hs*....25
   7.3. Litsents....27

1. Sissejuhatus

Andmebaasid on tänapäeval lahutamatu osa enamik tarkvarast, alati on vaja andmeid hoida ning neid välja anda ja uusi juurde lisada. Halb jõudlus andmebaasiga suhtlemisel võib oluliselt langetada tarkvara või teenuse kasutajakogemust ning kvaliteeti, mis omakorda võib mõjuda halvasti ka võimalikule kasumile. Sellepärast ongi oluline, et operatsioonid andmebaasidega toimuksid võimalikult kiiresti ning efektiivselt. Iga aastaga suureneb internetiühendusega seadmete arv, mis tähendab rohkem andmebaasidega suhtlemisi ning see ainult suurendab optimeerimise olulisust [1].

Mõned tüüpilised olukorrad, mille optimeerimine parandab andmebaasiga suhtlemise efektiivsust:

- Ükshaaval päringute tegemine andmebaasi
- Mitu korda identse päringu tegemine
- Eri andmebaasidest ükshaaval andmete pärimine, kui võiks seda korraga mitmest andmebaasist teha

Päringute optimeerimine võib olla arendajale väga tülikas ning aeganõudev, kuid on siiski sõltuvalt tarkavara või teenuse suurusest väga oluline. Sellepärast on loodud programmeerimiskeelte jaoks erinevaid teeke, mis ise optimeerivad päringuid, ilma et arendaja peaks seda tegema. *Haskell'*i jaoks on üheks selliseks teegiks *Haxl* [2]. Selle loojaks on *Facebook* ning teegi loojate sõnul lahendab see kõik eelnimetatud probleemid. Teek peaks päringuid tegema minimaalne arv kordi, ehk ebavajalikke korduvaid päringuid ei tehta ning kui võimalik, toimuvad kõik päringud paralleelselt, isegi erinevatesse andmete asukohtadesse. Kusjuures ei ole teek piiratud ainult andmebaasidele, seda teeki on võimalik kasutada igasuguste andmete asukohtadega, näiteks failisüsteemid, internetiteenused (nt *Facebook'*i API jt), kompilaatorid jt [3]. Lühidalt, kõike, mida on võimalik *Haskell'*is kujutada, on võimalik *Haxl* teegiga kasutada.

Käesolevas lõputöös uuritaksegi kahe näidisprogrammi näol, kas ja kuidas antud teek päringuid optimeerib. Loodud näidisprogrammid ja nende töötamine seletatakse lahti ning lisaks antakse teegist ka üldisem ülevaade. Tulemuseks on lõputöö, mille põhjal on võimalik lugejal otsustada teegi kasulikkuse üle.

Lõputöö esimeseks peatükiks on sissejuhatus. Teine peatükk annab ülevaate *Haxl* teegist ning kolmas on algelise näidisprogrammi kohta. Neljas peatükk on keerulisemast näidisprogrammist ning lisaks annab ülevaate HDBC teegist. Pärast seda tuleb kokkuvõte, ehk viies peatükk ning see-

järel viidatud kirjandus. Seitsmendaks peatükiks on lisad, kus on täispikkuses näidisprogrammide koodid ning litsents.

Loodud näidisprogrammide koodid leiab järgmiselt *GitHubi* URL-ilt https://github.com/Martiiii/haxl-naidis .

2. *Haxl* teek

Haxl [2] on *Haskell'*i teek, mis lihtsustab andmete pärimist andmebaasidest, veebirakendustest või muudest andmete asukohtadest.

Järgnevad lõigud põhinevad Marlow jt blogipostitusel [3] ning avaldatud artiklil [4]. *Haxl* avalikustati 10. juunil 2014 *Facebooki* poolt. See loodi nende spämmivastase süsteemi *Sigma* töö efektiivsemaks tegemiseks. Enne *Haxl'*it olid *Sigma* reeglid kirjutatud keeles FXL, kuid see nõudis liiga palju mälu ning protsessori ressursse. Sellepärast otsustatigi kasutusele võtta *Haskell* ning selle tulemusena loodigi *Haxl* (*Haskell* + FXL).

Teek oskab mitmest kohast paralleelselt andmeid pärida ja suudab ka mitu erinevat päringut samast allikast kokku põimida. Lisaks puhverdatakse päringuid, et üleliigseid päringute tegemisi andmete allikatest vähendada. *Haxl* otseselt ise ei tee päringuid, vaid optimeerib neid.

Haxl jooksutab kõik võimalikud päringud samaaegselt. See tähendab, kui on päring, mis koosneb kahest eraldiseisvast päringust, siis kõigepealt jooksutatakse korraga need kaks päringut. Pärast nende tulemuste kätte saamist jooksutatakse kolmas päring.

*Haxl'*i kasutamiseks on kõigepealt vaja luua kiht teegi ja andmete allikate vahele. *Haxl* teegis tähendab see andmeallikate implementeerimist. Kohad, kust andmed tulevad, ei ole piiratud, erinevaid allikaid saab defineerida erinevate tüüpidega. Iga allika jaoks on vaja implementeerida kohustuslikud *Haxl'*i meetodid ning ka funktsioonid, mis reaalselt teevad päringuid. Kui näiteks andmed asuvad andmebaasis, on vaja kasutada teeki, mis võimaldab teha päringuid andmebaasi.

Päringute puhverdamine vähendab ebavajalikke kordusi. Näiteks, kui samal ajal tehakse mitu identset päringut, siis *Haxl* teeb tegelikult ainult ühe, sest teised annaksid sama tulemuse. Päringu esmakordsel tegemisel andmeallikasse salvestatakse tulemus mällu, et samade päringute toimumisel saada tulemused mälust kiiremini kätte. Lisaks kindlustab see andmete ühesuse päringute tegemise ajal.

Tüübiohutuse jaoks võeti *Haskell'*is kasutusele *Typeable* klass ning GADT, ehk *Generalized algebraic datatypes. Haskell'*i *Applicative* ja *Functor* klassidest pärit operaatoritega (<*> ja *<$>)* muudeti funktsioonide jooksutamised asünkroonseteks.

3. Algeline näidisprogramm

3.1. Idee

Uuritava teegi tutvustamiseks ning seletamiseks luuakse kõigepealt algelisem näidisprogramm. Olgu näidisprogrammi aluseks andmebaas, kus hoitakse inimeste vanuseid. Arusaadavuse otstarbel tehakse näidisprogramm kujutletava andmebaasiga, kus on inimese kujutamiseks loodud ainult üks tabel nimega „Inimene" ning millel on vanuse kujutamiseks loodud üks väli nimega „Vanus".

Olgu olemasoleva andmebaasi jaoks ainult üks päring, mis tagastab ainult numbri, ehk vanuse. Järgnevas näitekoodis on näidatud, kuidas seda päringut jooksutada *Haxl* teeki kasutades. Eelmainitud päringut on näidisprogrammis välja kutsutud kaks korda, et demonstreerida ühte teegi funktsionaalsust. Selleks funktsionaalsuseks on päringute puhverdamine (i.k. *caching*). Päringut on tehtud topelt, kuid tegelikkuses jooksutatakse päringut ainult ühe korra, seejuures tagastatakse mõlemale päringu välja kutsumisele õige vastus. Päringu esmakordsel jooksutamisel salvestatakse selle tulemus mällu. Kui samal käivituskorral kutsutakse uuesti sama päring välja, võetakse tulemus hoopis mälust, mitte andmebaasist. See tagab andmete ühesuse erinevate päringute jooksutamise ajal ning on lisaks kiirem – ei tehta ebavajalikke päringuid andmebaasidesse.

Selle näidisprogrammi kogu kood on lisades (Lisa 1).

3.2. Päringute jooksutamine

```
paring :: IO ()
paring = do
    myEnv <- initEnv initialState ()
    r <- runHaxl myEnv (dataFetch Vanus)
    r1 <- runHaxl myEnv (dataFetch Vanus)
    print r
    print r1
```

Koodi selguse mõttes olgu päringute jooksutamise kood failis *NaideMain.hs*. Päringut tehakse funktsioonis *paring*, mis on tüüpi *IO ()*.

Intuitiivselt, tüüp *IO a* tähendab, et tegemist on tegevusega (i.k. *action*) *IO* monaadis, mis lõpuks annab tulemuseks *a*-tüüpi väärtuse. Tegevuste lõpuks, ehk viimaseks tegevuseks peab olema *a*-tüüpi väärtuse tagastamine. Järgnevalt on kasutatud *do*-notatsiooni, mis tähendab, et ridadel, kus on kasutatud operaatorit <-, tehakse kõigepealt paremal pool asuv tegevus ning selle tulemus salvestatakse vasakul olevasse muutujasse ning ridadel, kus on lihtsalt tegevused, jooksutatakse need tegevused

ilma midagi salvestamata või tagastamata. Operaatori *do* kasutamine teeb koodi paremini loetavaks, sest muidu peaks kasutama vastavalt vajadusele iga tegevuse järel >> või >>= operaatorit [5]. Kuna praeguses näites on tüübiks *IO ()*, tehakse lihtsalt erinevaid tegevusi, kuid ei tagastata midagi. Sellel teemal võib lugeda lähemalt ka Härmel Nestra raamatust [6] funktsionaalse programmeerimise kohta.

Kasutades funktsiooni *initEnv*, luuakse *paring* funktsioonis esmalt *Haxl*'i jaoks keskkond. Esimeseks argumendiks nõuab see funktsioon *StateStore*-tüüpi isendit. Teiseks argumendiks antakse sisendiks mingisugune u-tüüpi väärtus, mida võib päringute töötlemise ajal vaja minna. Funktsioon *initEnv* tagastab *IO (Env u)*-tüüpi väljundi.

Järgmiseks on rida, mis jooksutab päringu. Selleks kasutatakse *Haxl*'i funktsiooni *runHaxl*, mis esimeseks argumendiks võtab sisse eelnevalt loodud keskkonna *myEnv*. Teiseks argumendiks on oodatud *GenHaxl u a*-tüüpi isendit. *GenHaxl u* on monaad, kus toimub *Haxl* teegi töö. Sellist tüüpi isend saadakse funktsioonist, mis pärib küsitud andmeid andmeallikast ehk i.k. *Data Source*'ist. Selleks funktsiooniks on *dataFetch*, mis ainsaks argumendiks võtabki päringu tüübi, mida jooksutada (praeguse näitekoodi puhul *Vanus*). Funktsiooni *dataFetch* väljund sõltub *fetch* funktsiooni implementatsioonist, mis määratakse andmeallika loomisel. Funktsiooni *runHaxl* väljundiks ongi tehtud päringu tulemus. Järgmine rida on sama päringu jooksutamine, aga tulemus salvestatakse uude muutujasse. Viimasena kuvatakse näitekoodis veel päringute tulemused, kust selgub, et mõlema päringu tulemused on identsed.

```
initialState :: StateStore
initialState = stateSet NoState stateEmpty
```

See on mugavusfunktsioon, mis loob andmetüübi *Inimene* jaoks tühja oleku. Seda tehakse funktsiooniga *stateSet*, mille signatuuriks on *forall f. StateKey f => State f -> StateStore -> StateStore*. Funktsiooni esimeseks argumendiks on informatsioon, mida tahetakse olekuna hoida ning teiseks on *StateStore*-tüüpi isend, kuhu olek juurde lisatakse. Kuna hetkel see on esimene ja ainus olek, siis teiseks argumendiks on *stateEmpty*, mis kujutabki ilma olekuteta *StateStore*-i. Hoitav informatsioon (selles näidises *NoState*) peab olema defineeritud ka andmeallikas endas olekuna, seda vaadatakse hiljem. Funktsiooni väljundiks on *StateStore*-tüüpi isendi.

3.3. Andmeallika loomine

*Haxl'*i kasutamiseks eelnevalt nähtud viisil tuleb esmalt luua kiht *Haxl* teegi ja andmeallika vahel. Näitekoodi andmeallikaks on jätkuvalt kujutletav andmebaas, kus on ainult tabel *Inimene*, millel on üks väli *Vanus*. Et kood oleks selgem, tehakse see eraldi faili nimega *NaideDataSource.hs.* Järgnevad näiteprogrammi osad ei moodusta täieliku näidisprogrammi, ära on jäetud triviaalsed teekide importimised ning keelelaiendused. Täielikul kujul on näidisprogrammid nähtaval lõputöö lisades.

Kõigepealt luuakse andmetüüp kasutatava kujutletava andmebaasi tabeli *Inimene* jaoks, millel on üks konstruktor *Vanus* tüübiga *Inimene Int*. Sellisel kujul konstruktorite tüüpide määramiseks on vaja kasutada keelelaiendust *GADTs*. Keelelaienduse lisamist programmi saab näha lisadest (Lisa 1).

```
data Inimene a where
    Vanus :: Inimene Int
    deriving Typeable

deriving instance Show (Inimene a)
deriving instance Eq (Inimene a)
```

Loodud andmetüüp peab olema veel *Typeable*, *Show* ning *Eq* tüübiklassist. *Typeable* läheb *Haxl'il* vaja, et oleks võimalik turvaliselt tüübiteisendust teha (i.k. *casting*). Järgnevad andmeallika jaoks vajalikud klasside implementeerimised on pärit *Haxl'*i teegist, välja arvatud *Hashable.*

```
instance ShowP Inimene where
    showp = show
```

ShowP on *Haxl'*i klass andmetüübi konstruktorite parameetrite kuvamiseks. Selleks eksisteerib ainult üks funktsioon, *showp*. Koodijupi teise reaga määratakse *showp* funktsioon *Haskell'*i enda *show* funktsiooniga. Seda on võimalik teha, sest andmetüüp *Inimene* pärib *Show* klassi. Siinkohal on võimalik defineerida *showp* funktsiooni kuidasiganes kasutajal vaja, kuid praeguses näidisprogrammis sobib siin tavaline *Haskell'i show* funktsioon.

```
instance DataSourceName Inimene where
    dataSourceName _ = "Inimene"
```

*Haxl'*i andmeallikatele peab olema ka nimi ning seda määratakse defineerides *dataSourceName* funktsiooni *DataSourceName* klassist. Seda kasutatakse *Haxl*'i-siseselt andmete jälituses ning statistikas.

```
instance Hashable (Inimene a) where
    hashWithSalt salt Vanus = salt
```

Hashable klass laseb muuta mingi väärtuse räsiks, mis on täisarvuline väärtus. Näitekoodi implementeerimise jaoks on vaja määrata, kuidas funktsioon *hashWithSalt* käitub loodud konstruktoriga *Vanus*. Lihtsuse huvides selle väärtuseks määratud *salt*, sest näidises on ainult üks konstruktor. See klass on vajalik *Haxl'*i vahemälu jaoks, selle abil suudab teek aru saada, millised päringud on juba tehtud..

```
instance StateKey Inimene where
    data State Inimene = NoState
```

StateKey implementeerimine laseb igal andmeallikal hoida enda tüübiga päringutes informatsiooni ning selle implementeerimise jaoks läheb vaja *Typeable* klassi, mis sai *Inimene* andmetüübi loomisel sisse toodud. Andmebaasiga suhtlemise korral saaks siin hoida näiteks andmebaasiühendust, kuid kuna hetkel on tegemist lihtsa kujutletava andmebaasiga, siis pole vaja siin midagi hoida ning seetõttu on selle väärtus ebaoluline. Eelmises peatükis näidatud olek, *NoState*, tuleb siin andmeallika jaoks defineerida. Andmeallikale saab ainult neid olekuid määrata, mis on andmeallikas eelnevalt defineeritud.

```
instance DataSource () Inimene where
    fetch _ _ _ reqs = SyncFetch $ do
        forM_ reqs $ \(BlockedFetch req var) -> getValue req var
```

Selle klassi implementeerimisega muutub *Inimene* tüüp täielikult *Haxl*'i jaoks andmeallikaks. Sellel klassil on vaja määrata funktsioon *fetch*, see on põhifunktsioon, mis viib päringud läbi.

Funktsiooni *fetch* signatuur on *State req -> Flags -> u -> [BlockedFetch req] -> PerformFetch*. Sellest lähtudes on näha, et *fetch* võtab esimeseks argumendiks *State req* tüüpi väärtuse, kus *req* on selle andmeallika puhul *Inimene*. Kuna praeguses näidisprogrammis ei hoita selles midagi vajalikku (ainult eelnevalt defineeritud *NoState*), ei ole see argument oluline. Seetõttu pannakse esimese argumendi asemele alakriips. Teiseks argumendiks on *Flags* tüüpi väärtus. Nendega on võimalik kontrollida mõningaid *Haxl'i* seadeid, näiteks andmete jälitust ja puhverdamist. Neid selles näidisprogrammis vaja muuta ei ole, seega pannakse jällegi alakriips. Kolmas argument on *u*-tüüpi ning praeguse näite puhul ei kasutata ka seda, seega pannakse alakriips. Neljas argument on järjend *BlockedFetch req* tüüpi isenditest. *BlockedFetch req* sisaldab endas kahte asja – esimeseks on päringu tüüp, mida jooksutada (praegusel juhul *Vanus*) ja teiseks on koht, kuhu selle päringu tulemus panna. Selleks kohaks on *ResultVar* tüüpi isend. Sinna läheb päringu tulemus, olgu see erind või reaalne tulemus.

Funktsiooni *fetch* väljund peab olema *PerformFetch* tüüpi. Selle saavutamiseks on *Haxl*'ist kasutada kahte konstruktorit – *SyncFetch (IO ())* ja *AsyncFetch (IO () -> IO ())*. Teist kasutatakse tavaliselt juhul kui tahetakse päringute tegemise ajal paralleelselt midagi muud teha.

Kuna *SyncFetch* võtab argumendiks *IO ()*, luuakse *do*-plokk. Praeguses näidiskoodis on võimalik üldse teha ainult ühte päringut *Vanus*, seega funktsiooni *fetch BlockedFetch req* järjendi elemendid saavad ainult olla kõik *Vanus* konstruktoriga. Seda teades tagastatakse igale päringule küsitud vanus, ehk igale listi argumendile rakendatakse funktsiooni, mis tagastab vanuse. Selle saavutamiseks kasutatakse funktsiooni *forM_*, selle esimene argument on järjend ja teine argument on funktsioon, mida rakendatakse igale järjendi elemendile. Praegusel juhul läheb esimeseks argumendiks *BlockedFetch req* järjend ning teiseks argumentiks luuakse anonüümse funktsioon, mis võtab sisendiks *BlockedFetch* tüüpi isendi ning annab selle sisu edasi *getValue* funktsiooni. Nagu eelnevalt mainitud, on praegu ainult ühte tüüpi päringut ning sellepärast ei ole vaja kontrollida *BlockedFetch* päringu tüüpi. Kui programmis oleks mitu erinevat päringut, siis siin peaks kontrollima iga *BlockedFetch* tüüpi, et teada saada, mis päringut tehakse ja mida vastu oodatakse.

```
getValue :: Inimene a -> ResultVar a -> IO ()
getValue Vanus var = putSuccess var 1
```

See on funktsioon, mis annab igale päringule oodatud tulemuse. Esimene argument on konstruktor *Vanus*, selle järgi veendutakse, et teatud tüüpi päringule antakse temale mõeldud tüübiga vastus, aga kuna praegu eksisteerib ainult *Vanus* päringud, siis saavad kõik tehtavad päringud sellelt funktsioonilt vastuse. Teine argument on *ResultVar*-tüüpi isend, see on igas *BlockedFetch*-tüüpi isendis olemas, sinna sisestatakse päringu tulemus. Funktsioon *getVanus* tagastabki päringule küsitud vanuse, pannes etteantud *ResultVar*'i tulemuse. Kuna praegu on tegemist algelise näitega, siis iga päring saab edukalt vastuseks arvu, ehk vanuse. Päringu edukat täitmist tehakse *Haxl*'i funktsiooniga *putSuccess*, mis võtab esimeseks argumendiks *ResultVar*'i kuhu tulemus panna, ning teiseks argumendiks tulemuse väärtuse.

Kogu eelnevate kooditükkide tulemus on nüüd üks *Haxl'*i andmeallikas, kuhu saab teeki kasutades päringuid teha. Koodijupid kokku pannes saadaksegi kood, mis liidestab andmete allikad (andmebaasid, internetiteenused jm) ja *Haxl*'i.

4. Keerulisem näidisprogramm

4.1. Idee

Selles näidisprogrammis tehakse keerulisem päring reaalsesse lokaalsesse andmebaasi ning lisaks kasutatakse eelmises peatükis loodud andmeallikat. Andmebaas on tehtud *Sqlite3* peale [7]. Andmebaasiga suhtlemiseks kasutatakse HDBC teeki, millele on lisatud vastavad *Sqlite3* draiverid [8] [9]. Töökäigu lihtsustamiseks kasutatakse *Chinook'*i näidisandmebaasi [10]. See kujutab muusikapoodi, kus on tabelid artistide, albumite, laulude, arvete ja klientide jaoks. Selles näidisprogrammis kasutatakse andmebaasist [11] kahte tabelit - *Track* ja *Album*, ehk vastavalt muusikapala ja album. *Album* tabelil on kolm veergu - *AlbumId*, *Title* ja *ArtistId*. See tabel kujutab muusikaalbumit ning igaühe kohta hoitaksegi selle nimi, artisti ID ning albumi enda unikaalne ID. *Track* tabel kujutab muusikapala ning sisaldab mitut veergu: *TrackId*, *Name*, *AlbumId*, *MediaTypeId*, *GenreId*, *Composer*, *Milliseconds*, *Bytes*, *UnitPrice*. Selles näiteprogrammis kasutatakse *Track* tabelist ainult *AlbumId* ja *GenreId* veerge.

Näidisprogrammis tehakse andmebaasile kokku kaks päringut. Üheks päringuks on albumi ID küsimine albumi nime järgi ja teiseks päringuks on muusikapala nime küsimine žanri ID ja albumi ID järgi. Selle päringu sisenditeks on eelmise päringu tulemus ning lisaks veel eelmises peatükis loodud kujutletava andmebaasi ainsa päringu tulemus. Teisisõnu on see päring, mille sees on omakorda kaks eraldi päringut. Täpne päring, mis selles näidisprogrammis tehakse, on järjendi tagastamine muusikapaladest, mis on pärit albumist nimega „*For Those About To Rock We Salute You*" ning mille žanri unikaalse ID väärtuseks on eelmises peatükis loodud andmeallika vanuse küsimise päringu tulemus. Selline päring toob välja *Haxl'*i paralleelsuse – sisemised kaks päringut tehakse samal ajal ning alles siis kui on mõlema tulemused käes, on võimalik teha kolmas päring. Lisaks saab näha, et erinevatest andmeallikast pärit päringuid on võimalik kombineerida omavahel ning *Haxl* suudab sellega automaatselt toime tulla ilma kasutaja poolse lisatööta.

Ka selle näidisprogrammi kogu kood on lisades (Lisa 2).

4.2. *Haskell Database Connectivity* teek

Haskell Database Connectivity [8], ehk lühidalt HDBC on *Haskell'*is üks teekidest, millega on võimalik luua andmebaasidega ühendusi ning siis päringuid teha. Teegi abil kirjutatud päringud ja üldine kood töötab iga SQL andmebaasiga, millel on loodud draiver HDBC jaoks. Hetkel on võimalik saada neid *PostgreSQL*, *SQLite3*, *ODBC*, *MySQL* ning *Oracle* jaoks, ehk sõltuvalt kasutatavast andmebaasist, tuleb lisaks HDBC teegile paigaldada ka vastav draiver [12].

Lühidalt mõnest teegi funktsioonist [13]:

- Parameetritega päringud
- *Haskell'*i ja SQL tüüpidevaheline teisendus
- Andmebaasi metaandmete pärimine
- Andmebaasi serveri sätete pärimine

Parameetritega päringud kujutavad endast põhimõtteliselt SQL lauseid, kus teatud osasse saab panna muutuja. *SELECT* lausete puhul saab seda kasutada *WHERE* osas ning *INSERT* lausete puhul *VALUES* osas. See võimaldab teha mitut erinevat päringut andmebaasidesse kasutades nende jaoks ühte ja sama koodi, aga erineva parameetriga. HDBC teek võimaldab ka *Haskell'*i tüübid teisendada vastavateks SQL tüüpideks ning ka vastupidi. Kummagi operatsiooni jaoks on eraldi funktsioon. Teegiga on võimalik ka pärida infot andmebaasi enda kohta, näiteks tabelite või veergude nimed. HDBC teegi (ja ka üleüldise *Haskelli*) kasutamisest saab lugeda lähemalt Bryan O'Sullivani jt raamatust [14].

4.3. Päringute jooksutamine

Koodi selguse säilitamiseks on päringute jooksutamine tehtud eraldi faili *StoreMain.hs*. Nagu ka eelmises näidisprogrammis, käib ka siin kogu päringute jooksutamine *IO ()* tüüpi funktsioonis ning selle sisse tuleb ka siin esmalt luua keskkond, kus *Haxl* päringuid teeb. See luuakse jällegi funktsiooniga *initEnv*, kuid kuna nüüd on lisaks tegemist reaalse andmebaasiga, tuleb luua ka selle andmebaasiga ühendus. Eelmises näites *Haxl'*i andmeallikale ühtegi sisukat olekut ei antud, kuid nüüd oleks mõistlik olekuks luua seesama andmebaasiühendus, sest seda kasutatakse iga kord, kui tehakse päring andmebaasi. Funktsiooniga *initConnState* luuakse muutujasse *connState* see ühendus ning määratakse andmeallika olekuks. Selle funktsiooni sisu vaadatakse hiljem, andmeallika loomise juures. Niisiis funktsiooniga *initEnv* lisatakse *NoState* ja *connState* olekud loodavasse *Haxl'*i keskonda.

Järgmiseks tuleb päringu jooksutamine funktsiooniga *runHaxl*. Kui tahta teha mitmeid päringuid ning päringuid, mis sõltuvad mingist teisest päringust, siis tuleb need võrreldes algelisema näitega veidi teistmoodi kirja panna. Koodi arusaadavuse säilitamiseks on loodud päringute funktsioonide jaoks eraldi fail, *StoreHaxl.hs*. Funktsioon *getTrackAlbumGenre* on muusikapala nime küsimine albumi ID ja žanri ID järgi, ehk võtab argumendiks funktsioonide *getAlbumId* ning *getVanusNr* tulemused. Esimene nendest võtab argumendiks sõne, mis *getAlbumId* puhul kujutab albumi nime. Funktsioon *getVanusNr* on eelnevalt loodud kujutletavast andmebaasist vanuse küsimise päring.

```
paring :: IO ()
paring = do
  connState <- initConnState
  myEnv <- initEnv (stateSet NoState $ stateSet connState $ stateEmpty) ()
  r <- runHaxl myEnv $ getTrackAlbumGenre
    (getAlbumId "For Those About To Rock We Salute You")
    getVanusNr
  print r
```

Algelises näitekoodis oli ainult üks päring ilma ühegi argumendita. Praeguses koodis on kolm päringut, millest kaks võtavad vähemalt ühe argumendi sisendiks. Kui päringud oleksid samamoodi lihtsalt *do*-plokis *runHaxl* funktsioonidega üksteise järgi, nagu algelises näidisprogrammis, siis tehakse nad kõik ükshaaval. Paralleelsuse puudumine tuleneb sellest, et *do*-plokis tehakse rangelt kõik tegevused järjest, iga järgnev rida tehakse alles siis, kui eelmine on lõpetanud. Paralleelsuse saamiseks ei ole vaja palju muudatusi teha, kõik päringud tuleb jooksutada ühes *runHaxl* funktsioonis. Nagu viimasest koodilõigust näha, ei ole see iga päringu jaoks loodud funktsioonidega eriti keeruline kuju, *runHaxl* funktsiooni viimaseks argumendiks on eelnevalt mainitud päringu funktsioon, mis võtab enda argumentideks kahe teise päringu funktsioonid, millest üks võtab omakorda sisendiks sõne. Viimane rida selles koodilõigus kuvab päringu tulemuse, milleks on *["For Those About To Rock (We Salute You)","Put The Finger On You","Let's Get It Up","Inject The Venom","Snowballed","Evil Walks","C.O.D.","Breaking The Rules","Night Of The Long Knives","Spellbound"]*. See on nimekiri muusikapaladest, mis on pärit albumist nimega „*For Those About To Rock We Salute You*” ning mille žanrideks on *Rock*, sest selle unikaalseks ID-ks andmebaasis on *1*, ehk vanuse küsimise päringu tulemus.

SQL keeles saab sõltuvaid päringuid ühendada JOIN konstruktsiooniga nii, et vastavad alampäringud ning nende ühendamine toimub ühe korraga andmebaasiserveris [15]. *Haxl* sellist optimeerimist ei võimalda. Niisiis annab *Haxl* suurima eelise just mitte-SQL andmeallikatega töötamisel.

Järgmises koodilõigus näidatakse, kuidas peavad päringute jooksutamise lihtsustamise jaoks loodud funktsioonid kirjas olema, et kasutada ära *Haxl'*i paralleelsus ning päringute kombineerimine.

```
getTrackAlbumGenre :: GenHaxl () Int -> GenHaxl () Int -> GenHaxl () [String]
getTrackAlbumGenre albumId artist =
    dataFetch =<< GetTrackByAlbumGenre <$> albumId <*> artist

getAlbumId :: String -> GenHaxl () Int
getAlbumId albumName = dataFetch =<< GetAlbumId <$> (pure albumName)

getVanusNr :: GenHaxl () Int
getVanusNr = dataFetch =<< (pure Vanus)
```

Kuna *GenHaxl* monaad kasutab ära *Haskell*'i *Applicative* ja *Functor* tüübiklasse, saavutatakse paralleelsus kasutades vastavalt <*> ja <$> operaatoreid.

Operaator <*> tuleb *Applicative* tüübiklassist ning selle signatuuriks *Haxl'*is on *GenHaxl u (a -> b) -> GenHaxl u a -> GenHaxl u b*. See funktsioon võtab sisendiks kaks argumenti, esimeseks on *GenHaxl u*-tüüpi funktsioon, mis võtab sisse *a*-tüüpi väärtuse ja tagastab *b*-tüüpi väärtuse ning teiseks argumendiks on *GenHaxl u a*-tüüpi väärtus. Funktsioon <*> tagastab *GenHaxl u b*-tüüpi väärtuse.

Operaator <$> tuleb *Functor* tüübiklassist, see on *infix* vorm funktsioonist *fmap* ning selle signatuuriks on *(a -> b) -> GenHaxl u a -> GenHaxl u b*. See funktsioon võtab esimeseks argumendiks funktsiooni, mis võtab sisse *a*-tüüpi väärtuse ning tagastab *b*-tüüpi väärtuse. Teiseks argumendiks võtab *GenHaxl u a*-tüüpi väärtuse ning tagastab *GenHaxl u b*-tüüpi väärtuse.

Operaatori =<< signatuuriks on *(a -> GenHaxl u b) -> GenHaxl u a -> GenHaxl u b*. Selle funktsiooni esimeseks argumendiks on funktsioon, mis võtab sisse *a*-tüüpi väärtuse ning tagastab *GenHaxl u b*-tüüpi väärtuse. Teiseks argumendiks on *GenHaxl u a*-tüüpi väärtus ning funktsioon tagastab *GenHaxl u b*-tüüpi väärtuse. Teisisõnu, teisest argumendist tulnud väärtus antakse esimese argumendi funktsiooni sisendiks ning saadakse sellest *b*-tüüpi väärtus. See funktsioon on sama nagu monaadi *bind* operaator >>=, aga argumendid on omavahel vahetatud.

Funktsioon *pure* signatuuriks on *a -> GenHaxl u a*, ehk see toob lihtsalt *a*-tüüpi väärtuse *GenHaxl u* monaadi.

Andmeallikast pärit päringu tüüp ja selle argumendid tuleb eraldada <$> operaatoriga ning argumendid tuleb omavahel eraldada <*> operaatoriga. Näiteks *GetTrackByAlbumGenre <$> albumId <*> artist*. Kui sellises päringus on *albumId* ning *artist* omakorda päringud, tehakse kõigepealt paralleelselt need päringud ning nende tulemused antakse *GetTrackByAlbumGenre* päringule sisendiks, pärast mida tehakse see päring. Funktsioon *dataFetch* võtab sisendiks andmeallika tüübi või nende kombinatsiooni ning seejärel teostab päringud *GenHaxl* monaadis. Sisend tuleb =<< operaatorist. Teisisõnu, =<< operaatorist paremal pool olev osa valmistab päringu tegemise ette (kas on sõltuv või mitte) ning vasak pool, ehk *dataFetch*, teostab selle päringu.

Funktsioonide *getAlbumId* ja *getGenre* puhul tuleb sisenditeks saadud *String* tüüpi väärtused teisendada *GenHaxl () String* tüüpi kasutades funktsiooni *pure*, sest kogu päringute optimeerimise loogika käib *GenHaxl* monaadis.

4.4. Andmeallikate loomine

Andmeallikas on koodi arusaadavuse säilitamiseks tehtud failil *StoreDataSource.hs*. Nagu eelmise näiteprogrammi juures näha oli, tuleb ka praegu teha kasutatav andmebaas *Haxl'*i jaoks andmeallikaks. Enamasti on kood ja implementeeritavad funktsioonid analoogsed.

Esmalt tuleb luua andmetüüp ning kuna praeguses näites on kolm päringut, millest üks on *Inimene* andmeallikast, tuleb ka andmetüübile iga päringu kohta enda konstruktor teha ja vastavad tüübid argumentide ning väljundi jaoks.

```
data Store a where
    GetTrackByAlbumGenre  :: Int -> Int -> Store [String]
    GetAlbumId            :: String -> Store Int
    deriving Typeable

deriving instance Show (Store a)
deriving instance Eq (Store a)


instance DataSourceName Store where
    dataSourceName _ = "TheStore1"
instance ShowP Store where
    showp = show

instance Hashable (Store a) where
    hashWithSalt salt (GetTrackByAlbumGenre a b) = 2 * (a + b + salt)
    hashWithSalt salt (GetAlbumId a) = 2 * ((length a) + salt) + 1

instance StateKey Store where
    data State Store = ConnState {connection :: Connection}
```

Samamoodi tuleb implementeerida *Show*, *Eq*, *dataSourceName*, *showp* ja *hashWithSalt*. Esimene suurem erinevus kujutletava ning reaalse andmebaasi vahel tuleb andmeallikale oleku loomise juures. Nagu eelnevalt mainitud, on iga päringuga kaasas tema olek (*State*) ning iga päringu tegemise juures on vaja andmebaasiga suhelda. Sellest võib järeldada, et mõistlik oleks luua andmebaasi ühendus andmeallika olekusse. Implementeeritava andmetüübi *State Store* ainsaks konstruktoriks ongi *connection*, mis hoiab endas *Connection*-tüüpi andmebaasiühendust.

Järgmine suurem erinevus *Haxl*'i andmeallika loomisel tuleb *fetch* funktsiooni implementeerimisel, sest nüüd on rohkem kui üks päring ning tegemist on ka reaalse andmebaasiga.

```
instance DataSource () Store where
    fetch (ConnState db) _ _ reqs = SyncFetch $ do
        forM_ reqs $ \(BlockedFetch req var) -> runQuery db req var
```

Funktsiooni *fetch* esimene argument on vajaminev andmebaasiühendus, seega ei märgita seda alakriipsuga. Kuna praegu on andmeallikal mitut tüüpi päringuid, ei saa siinkohal kohe otse tulemust tagastada. Iga päringu vajalikud osad (andmebaasiühendus, päringu tüüp ning tulemuse hoidja) antakse funktsioonile *runQuery* sisendiks.

```
runQuery :: Connection -> Store a -> ResultVar a -> IO ()
runQuery db (GetTrackByAlbumGenre x y) var =
    getTrackByAlbumGenre x y db var
runQuery db (GetAlbumId x) var = getAlbumsId x db var
```

Kasutades mustrisobitust (i.k. *pattern matching*) [16] saame iga päringu tüübi kohta tagastada sellele vastava väärtuse. Siin ongi niiöelda ühenduskoht *Haxl'*i ja andmete kättesaamise funktsioonide vahel. Näites olev *runQuery* funktsiooni tagastab igale päringu tüübile temale vastava HDBC päringu tulemuse.

```
initConnState :: IO (State Store)
initConnState = do
    conn <- connectSqlite3 "test2.db"
    return ConnState {connection = conn}
```

Funktsioon *initConnState*-ga sai loodud ühendus andmebaasiga ning selle väljund andmeallika olekuks määratud. Nagu siit koodijupist näha, luuakse funktsioonis esmalt ühendus andmebaasiga *test2.db* kasutades HDBC funktsiooni *connectSqlite3*. See ongi osa, mida läheb iga päringu juures andmebaasiga suhtlemiseks vaja. Kogu funktsioon tagastabki selle ühenduse eelnevalt defineeritud *State Store* tüübina, ainsale konstruktorile *connection* väärtustatakse seesama andmebaasiühendus.

Järgmisena tulevadki funktsioonid, mis kasutades HDBC teeki, suhtlevad andmebaasiga ning pärivad küsitavad andmed. Esimeseks on funktsioon *getTrackByAlbumGenre*, mis tagastas järjendi laulunimedest, mis on sisendalbumist ning -žanrist.

```
getTrackByAlbumGenre :: Int -> Int -> Connection ->
    ResultVar [String] -> IO ()
getTrackByAlbumGenre albumId genre db var =
    do
        r <- quickQuery' db
            "SELECT Name FROM Track WHERE AlbumId = ? AND GenreId = ?;"
            [toSql albumId, toSql genre]
        let strings = map convRow r
        putSuccess var strings

    where convRow :: [SqlValue] -> String
          convRow [sqlName] = name
              where name = (fromSql sqlName)::String
          convRow x = error $ "Unexpected result: " ++ show x
```

Esimesed kaks argumenti on, nagu eelnevalt mainitud, albumi ID ning žanri ID. Kolmandaks argumendiks on andmebaasiühendus, mida läheb vaja andmebaasiga suhtlemiseks ning viimaseks argumendiks on *Haxl'*i muutuja, kuhu salvestatakse päringu tulemus.

do-plokis esimene asi on päring andmebaasi, see tehakse HDBC funktsiooniga *quickQuery'*. Funktsioon võtab sisendiks andmebaasiühenduse, *SELECT*-lause ning päringu filtreerimiseks sisendparameetrite järjendi. HDBC teek teisendab *Haskell*'i tüübid automaatselt õigeteks SQL tüüpideks [17] funktsiooniga *toSql* ning vastupidise operatsiooni jaoks funktsiooniga *fromSql.* Päringu tulemuseks on järjend, kus iga element on omakorda järjend, mille sees on tingimustele vastav laulu nimi ning päringu tulemus salvestatakse muutujasse *r*. Järgmisena teisendatakse saadud tulemus vajalikule kujule (selles näites *[String]*). See saavutatakse funktsiooniga *map*, mis rakendab igale muutuja *r* elemendile funktsiooni *convRow*. Viimane võtab sisendiks *[SqlValue]*-tüüpi isendi ning tagastab *String*-tüüpi isendi. Esmalt teisendab *fromSql* funktsiooniga SQL-tüüpi väärtuse *Haskell'*i *String* tüübiks ning siis tõstab selle järjendist välja. Tehes neid operatsioone iga muutja *r* elemendiga, tulebki muutjasse *strings* järjend *String*-tüüpi elementidest, ehk laulu nimedest. Viimaseks tulebki *Haxl'*ile teada anda, et saadud tulemus on õige. Selleks kasutatakse funktsiooni *putSuccess*, mis paneb talle sisendiks antud *ResultVar a*-tüüpi muutujasse teiseks sisendiks antud väärtuse. Kui on võimalus, et päring võib ebaõnnestuda, tuleb see variant lõpetada funktsiooniga *putFailure*, et *Haxl'*ile operatsiooni ebaõnnestumisest teada anda.

Funktsioonile *putSuccess* sisendiks antud *ResultVar a* tüüpi muutuja ongi seesama konteiner, mis tuleb iga *BlockedFetch* päringuga kaasa.

Järgmise päringu funktsiooni on analoogne, ainsad erinevused tulevad sisse SQL-tüüpi väärtuste *Haskell'*i väärtusteks teisendamises.

```
getAlbumsId :: String -> Connection -> ResultVar Int -> IO ()
getAlbumsId albumName db var =
    do
        r <- quickQuery' db
            "SELECT AlbumId FROM Album WHERE Title = ?;"
            [toSql albumName]
        let strings = map convRow r
        putSuccess var (head strings)

    where convRow :: [SqlValue] -> Int
          convRow [sqlName] = name
              where name = (fromSql sqlName)::Int
          convRow x = error $ "Unexpected result: " ++ show x
```

Funktsioon *getAlbumsId* võtab sisendiks albumi nime ning tagastab tema ID. Kuna *getAlbumsId* väljund on *Int*-tüüpi, siis funktsioonis *convRow* tuleb saadud SQL väärtus teisendada *Haskell'*i *Int* tüübiks.

Nagu eelnevatest koodijuppidest näha, siis *Haxl'*i kasutamine on suures osas iga andmeallikaga samasugune, erinevused tulevadki sisse ainult andmeallikate loomises. Tuleb osata ühendada *Haxl* ning misiganes muu teek, või ühendus, mida kasutatakse.

5. Kokkuvõte

Käesolevas lõputöös tutvustati ning demonstreeriti päringute optimeerimise teeki *Haskell'*is, mille nimeks on *Haxl*. Teegi kasutamine, põhilised funktsionaalsused ning nende töötamise loogika toodi välja luues kaks näidisprogrammi. Esimene nendest loodi kujutletavale andmebaasile, et teegi töökäigu arusaamist lihtsustada, kuid teise näidisprogrammi juures kasutati reaalset lokaalset andmebaasi. Lisaks *Haxl'*ile, kasutati teises näidisprogrammis andmebaasiga suhtlemiseks *Haskell Database Connectivity* teeki. Teine näidisprogramm näitas ning seletas lahti teegi keerulisemad optimeerimise osad ning nende kasutamise andmebaasiga.

Lisaks on lõputöös antud *Haxl'*i kohta üldisem ülevaade ning teegi tekkimise põhjus. Peale selle toodi välja olulisemad *Haskell'*i osad, mida teegi loojad *Haxl'*i jaoks kasutasid. Lühike ülevaade anti ka *Haskell Database Connectivity* teegi kohta.

Töö tulemuseks on kaks *Haxl* teeki tutvustavat näidisprogrammi ja nende seletused, mis demonstreerivad teegi kasutamist ning mille abil on võimalik lugejal otsustada teegi kasulikkuse ning keerukuse üle.

6. Viidatud kirjandus

[1] Turner V. The Digital Universe of Opportunities: Rich Data and the Increasing Value if the Internet of Things. 2014. https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm (28.04.2017)

[2] Hackage, Haxl. https://hackage.haskell.org/package/haxl-0.5.0.0 (28.04.2017)

[3] Marlow S, Purdy J. Open-Sourcing Haxl, a library for Haskell. 2014. https://code.facebook.com/posts/302060973291128/open-sourcing-haxl-a-library-for-haskell/ (28.04.2017)

[4] Marlow S, Brandy L, Coens J, Purdy J. There is no fork: An abstraction for efficient, concurrent, and concise data access. InACM SIGPLAN Notices 2014 Aug 19 (Vol. 49, No. 9, pp. 325-337). ACM.

[5] Haskell.org. A Gentle Introduction to Haskell: Input/Output. https://www.haskell.org/tutorial/io.html (28.04.2017)

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

[7] Sqlite, index. https://www.sqlite.org/index.html (28.04.2017)

[8] Hackage, HDBC. https://hackage.haskell.org/package/HDBC (28.04.2017)

[9] Hackage, HDBC-sqlite3. https://hackage.haskell.org/package/HDBC-sqlite3 (28.04.2017)

[10] Codeplex, Chinook Database. https://chinookdatabase.codeplex.com (28.04.2017)

[11] Codeplex, Chinook Database – Data Model https://chinookdatabase.codeplex.com/wikipage?title=Chinook_Schema&referringTitle=Documentation (28.04.2017)

[12] Github, HDBC wiki. https://github.com/hdbc/hdbc/wiki (28.04.2017)

[13] Github, HDBC wiki – Features. https://github.com/hdbc/hdbc/wiki/FeatureList (28.04.2017)

[14] O'Sullivan B, Stewart D, Goerzen J. Real World Haskell. O'Reilly Media. 2008

[15] Ullmann J, Widom J. A First Course in Database Systems. Prentice-Hall. 1997

[16] Lipovača M. Learn You a Haskell for Great Good!. Ljubljana: No Starch Press. 2011

[17] W3schools, SQL General Data Types.
https://www.w3schools.com/sql/sql_datatypes_general.asp (28.04.2017)

7. Lisad

7.1. Näidisprogramm 1

7.1.1. *NaideMain.hs*

```
module NaideMain where

import NaideDataSource
import Haxl.Core

paring :: IO ()
paring = do
    myEnv <- initEnv initialState ()
    r <- runHaxl myEnv (dataFetch Vanus)
    r1 <- runHaxl myEnv (dataFetch Vanus)
    print r
    print r1

initialState :: StateStore
initialState = stateSet NoState stateEmpty
```

7.1.2. NaideDataSource.hs

```
{-# LANGUAGE DeriveDataTypeable, GADTs, MultiParamTypeClasses,
OverloadedStrings, StandaloneDeriving, TypeFamilies #-}
module NaideDataSource where
import Control.Monad
import Data.Hashable
import Data.Typeable
import Haxl.Core
import Text.Printf

data Inimene a where
    Vanus :: Inimene Int
    deriving Typeable

deriving instance Show (Inimene a)
deriving instance Eq (Inimene a)

instance DataSourceName Inimene where
    dataSourceName _ = "Inimene"

instance ShowP Inimene where
    showp = show

instance Hashable (Inimene a) where
    hashWithSalt salt Vanus = salt
```

```
instance StateKey Inimene where
    data State Inimene = NoState

instance DataSource () Inimene where
    fetch _ _ _ reqs = SyncFetch $ do
        forM_ reqs $ \(BlockedFetch req var) -> getValue req var

getValue :: Inimene a -> ResultVar a -> IO ()
getValue Vanus var = putSuccess var 1
```

7.2. Näidisprogramm 2

7.2.1. *StoreMain.hs*

```
module StoreMain where

import StoreDataSource
import StoreHaxl
import NaideDataSource
import Control.Monad
import Database.HDBC.Sqlite3
import Database.HDBC
import Data.Hashable
import Data.Typeable
import Text.Printf
import Haxl.Core

paring :: IO ()
paring = do
  connState <- initConnState
  myEnv <- initEnv (stateSet NoState $ stateSet connState $ stateEmpty) ()
  r <- runHaxl myEnv $ getTrackAlbumGenre
    (getAlbumId "For Those About To Rock We Salute You")
    getVanusNr
  print r
```

7.2.2. *StoreHaxl.hs*

```
module StoreHaxl where

import Control.Monad
import StoreDataSource
import NaideDataSource
import Haxl.Core

getTrackAlbumGenre :: GenHaxl () Int -> GenHaxl () Int -> GenHaxl () [String]
getTrackAlbumGenre albumId artist =
    dataFetch =<< GetTrackByAlbumGenre <$> albumId <*> artist

getAlbumId :: String -> GenHaxl () Int
getAlbumId albumName = dataFetch =<< GetAlbumId <$> (pure albumName)

getVanusNr :: GenHaxl () Int
getVanusNr = dataFetch =<< (pure Vanus)
```

7.2.3. *StoreDataSource.hs*

```
{-# LANGUAGE DeriveDataTypeable, GADTs, MultiParamTypeClasses,
OverloadedStrings, StandaloneDeriving, TypeFamilies #-}
module StoreDataSource where
import Control.Monad
import Database.HDBC.Sqlite3
import Database.HDBC
import Data.Hashable
import Data.Typeable
import Text.Printf
import Haxl.Core

data Store a where
    GetTrackByAlbumGenre  :: Int -> Int -> Store [String]
    GetAlbumId            :: String -> Store Int
    deriving Typeable

deriving instance Show (Store a)
deriving instance Eq (Store a)

instance DataSourceName Store where
    dataSourceName _ = "TheStore1"

instance ShowP Store where
    showp = show
```

```
instance Hashable (Store a) where
    hashWithSalt salt (GetTrackByAlbumGenre a b) = 2 * (a + b + salt)
    hashWithSalt salt (GetAlbumId a) = 2 * ((length a) + salt) + 1

instance StateKey Store where
    data State Store = ConnState {connection :: Connection}

instance DataSource () Store where
    fetch (ConnState db) _ _ reqs = SyncFetch $ do
        forM_ reqs $ \(BlockedFetch req var) -> runQuery db req var

runQuery :: Connection -> Store a -> ResultVar a -> IO ()
runQuery db (GetTrackByAlbumGenre x y) var = getTrackByAlbumGenre x y db var
runQuery db (GetAlbumId x) var = getAlbumsId x db var

initConnState :: IO (State Store)
initConnState = do
    conn <- connectSqlite3 "test2.db"
    return ConnState {connection = conn}

getTrackByAlbumGenre :: Int -> Int -> Connection ->
    ResultVar [String] -> IO ()
getTrackByAlbumGenre albumId genre db var =
    do
        r <- quickQuery' db
            "SELECT Name FROM Track WHERE AlbumId = ? AND GenreId = ?;"
            [toSql albumId, toSql genre]
        let strings = map convRow r
        putSuccess var strings

    where convRow :: [SqlValue] -> String
          convRow [sqlName] = name
              where name = (fromSql sqlName)::String
          convRow x = error $ "Unexpected result: " ++ show x

getAlbumsId :: String -> Connection -> ResultVar Int -> IO ()
getAlbumsId albumName db var =
    do
        r <- quickQuery' db "SELECT AlbumId FROM Album WHERE Title = ?;"
          [toSql albumName]
        let strings = map convRow r
        putSuccess var (head strings)

    where convRow :: [SqlValue] -> Int
          convRow [sqlName] = name
              where name = (fromSql sqlName)::Int
          convRow x = error $ "Unexpected result: " ++ show x
```

7.3. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Marti Mutso**,
(*autori nimi*)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
Haxl teegi uurimine,
(*lõputöö pealkiri*)

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

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **30.04.2017**