TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Hele-Andra Kuulmets

Tüübiklassidega funktsionaalprogrammeerimine Scalas

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani, PhD

Tartu 2017

Tüübiklassidega funktsionaalprogrammeerimine Scalas

Lühikokkuvõte: Scala on Java virtuaalmasinat kasutav mitme-paradigma programmeerimiskeel, mis võimaldab omavahel kombineerida objektorienteeritud ja funktsionaalseid tehnikaid. Scala kui funktsionaalse keele miinuseks on, et selle standardteegist puuduvad funktsionaalprogrammeerijale harjumuspärased abstraktsioonid – tüübiklassid, millega saab kirjeldada erinevate tüüpide ühiseid omadusi, ilma et need peaksid pärinema samast ülemklassist. Selle puudujäägi kõrvaldamiseks on Scala jaoks loodud neid abstraktsioone sisaldav teek nimega Cats. Antud bakalaureusetöös uuriti, kuidas on viis Haskellist tuntud tüübiklassi Catsis realiseeritud ning milliste probleemide lahendamist need lihtsustavad. Selle tulemusena valminud töö kirjalik osa kujutab endast sissejuhatavat materjali tüübiklassidega funktsionaalprogrammeerimisest Scalas. Seda täiendavad harjutusülesanded ja näidisprogramm.

Võtmesõnad: funktsionaalprogrammeerimine, tüübiklassid, Scala

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

Functional programming with type classes in Scala

Abstract: Scala is a multi-paradigm programming language running on a Java virtual machine which allows to combine object-oriented and functional techniques. The disadvantage of Scala as a functional programming language is that its standard library is missing some useful abstractions – type classes which can be used to describe common properties of different types without the need to inherit them from the same superclass. To eliminate that disadvantage, a library named Cats containing those abstractions was created. The aim of this thesis is to research that library – study the implementations of five type classes that are already known from Haskell and describe the kind of problems they help to solve in Scala. The written part of this thesis provides an introduction to functional programming with type classes in Scala. In addition, practical exercises and an example program were written to complement the written part.

Keywords: functional programming, type classes, Scala

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

Sisukord

1 Sissejuhatus

Scala on Java virtuaalmasinat kasutav mitme-paradigma programmeerimiskeel, mis võimaldab kombineerida objektorienteeritud meetodeid funktsionaalprogrammeerimisega. Oma olemuselt puhtalt objektorienteeritud Scalas on samas olemas peaaegu kõik, mida ühelt funktsionaalselt keelelt oodata: funktsioonid on seal esimest klassi, eksisteerivad kõrgemat järku funktsioonid, mittemuteeritavad andmestruktuurid ja funktsionaalsed vahendid kõrvalefektide käsitlemiseks.

Samas puuduvad Scala standardteegist paljud puhtas funktsionaalprogrammeerimises kasutatavad abstraktsioonid, mille abil saaks kirjutada väga kasulikku geneerilist koodi. Typelevel.scala on proovinud seda puudujääki kõrvaldada, luues Scala jaoks neid abstraktsioone sisaldava teegi nimega Cats. Cats on lühend ingliskeelsest mõistest *category theory*, kust pärineb nende abstraktsioonide matemaatiline taust. Sarnaselt Haskellile on Catsis abstraktsioonid realiseeritud tüübiklassidena.

Käesoleva töö eesmärk on koostada ülevaade viiest põhilisest Catsi tüübiklassist ja uurida, mida kasulikku ning huvitavat nendega Scalas teha saab. Poolenisti referatiivse, poolenisti praktilise töö tulemusena on valminud sissejuhatustav materjal tüübiklassidega funktsionaalprogrammeerimisest Scalas, mida võib edaspidi kasutada õppematerjalina. Seda täiendavad praktilised harjutusülesanded vastavate tüübiklasside kohta ja üks väike Catsi tutvustav näiteprogramm. Harjutusülesandeid on tulevikus plaanis kasutada praktikumiülesannete ja näiteprogrammi loengumaterjalide koostamisel.

Töö teises peatüks seletatakse lahti funktsionaalprogrammeerimisega seotud põhitõed ja mõisted, mille tundmist töö lugejalt edaspidi eeldatakse. Kolmas peatükk on pühendatud konkreetselt tüübiklassidega seotud mõistete selgitamisele. Neljandas peatükis antaksegi ülevaade viiest põhilisest Catsi tüübiklassist. Iga tüübiklassi juures on toodud ka näiteid selle praktiliste kasutusviiside kohta. Viiendas peatükis kirjeldatakse lühidalt valminud näiteprogrammi ja harjutusülesandeid ning nende koostamise põhimõtteid. Samuti võib sealt leida programmeerimise õpetamisega seotud inimeste tagasiside neile ülesannetele. Lisadena on tööga kaasas koostatud harjutusülesanded ja näidisprogrammi lähtekood.

Töös on mitmes kohas mõistete ja tüübiklasside defineerimisel lisaks Scalale toodud näiteid ka puhtast funktsionaalsest keelest Haskell. Seda põhjusel, et töö autori arvates on mõningaid funktsionaalsete keelte konstruktsioone lihtsam õppida Haskelli põhjal.

2 Scala taustteadmised

Selles peatükis seletatakse lahti funktsionaalprogrammeerimisega seotud mõisted ja teemad, mille tundmist töö lugejalt edaspidi eeldatakse.

2.1 Funktsioonid funktsionaalprogrammeerimises

Funktsionaalprogrammeerimises mõistetakse funktsioonide all puhtaid funktsioone ehk funktsioone, millel puuduvad kõrvalefektid (sisend-väljund, erindid, oleku muutmine). See tähendab, et funktsioon ei mõjuta programmi täitmist muul viisil, kui sisendite põhjal vastuse arvutamisega. Funktsiooni tagastusväärtus sõltub ainuüksi funktsiooni sisenditest [CB15, lk 9].

Erinevalt imperatiivsest programmeerimisest on funktsionaalprogrammeerimises funktsioonid esimest klassi objektid (*first-class objects*), mis tähendab, et neid käsitletakse nagu kõiki teisi väärtusi: nad võivad olla teistele funktsioonidele nii argumentideks kui ka tagastusväärtuseks, neid võib omistada muutujatele ning hoiustada andmestruktuurides [Hasd].

2.2 Ilmutatud viidatavus

Avaldis e on ilmutatult viidatav (*referentially transparent*), kui iga programmi p puhul saab programmis p kõik avaldise e esinemised asendada tema väärtusega ilma programmi p tähendust mõjutamata. Funktsioon f on puhas funktsioon, kui avaldis *f(x)* on ilmutatult viidatav iga ilmutatult viidatava muutuja x puhul [CB15, lk 10].

Funktsiooni ilmutatud viidatavus tagab, et kõik, mida see funktsioon teeb, on välja loetav tema tagastusväärtusest [CB15, lk 11-12].

2.3 Rekursioon funktsionaalprogrammeerimises

Rekursiooni kasutatakse funktsionaalprogrammeerimises tsüklite kirjeldamiseks ilma muteeritava tsüklimuutujata. Tsükli hetkeolekut antakse edasi rekursiivse funktsiooni argumentidega. Kui funktsiooni kõik rekursiivsed väljakutsed on saba-rekursiivsed, siis Scala kompilaator kompileerimise ajal optimeerib selle funktsiooni automaatselt iteratiivseks tsükliks [CB15, lk 20].

2.4 Polümorfism

Monomorfne funktsioon on funktsioon, mis võtab argumendiks ja tagastab ainult ühest kindlast andmetüübist väärtusi.

Polümorfne (geneeriline) funktsioon on funktsioon, mis võtab argumendiks ja tagastab mistahes andmetüüpidest väärtusi. Näiteks:

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {
  def loop(n: Int): Int =
    if (n >= as.length) -1
    else if (p(as(n))) n
    else loop(n + 1)

  loop(0)
}
```

Funktsiooni nimele järgnev nurksulgudega ümbritsetud tüübiparameetrilist, antud näites `[A]`, sisaldab tüübiparameetreid, mida saab tüübimuutujatena ülejäänud funktsiooni signatuuris kasutada. Antud näites kasutatakse tüübimuutujat `A` kahes kohas: järjendi elemendid peavad olema tüüpi `A` ning funktsioon `p` võtab argumendiks elemendi tüüpi `A`, s.t järjendi elemendid ja funktsiooni `p` argument peavad alati olema samast tüübist [CB15, lk 23-24].

2.5 Kõrgemat järku funktsioonid

Kõrgemat järku funktsioonid on funktsioonid, mis võtavad argumendiks teisi funktsioone või tagastavad ise funktsioone [Doce]. Eelmises lõigus toodud funktsioon `findFirst` on näide kõrgemat järku funktsioonist. Kõrgemat järku funktsioonide välja kutsumisel on sageli mugavam neile argumendiks anda anonüümseid funktsioone (funktsiooniliteraale), selle asemel, et eraldi nimeline funktsioon defineerida ainult selleks, et seda teisele funktsioonile argumendiks anda.

Süntaktiliselt koosneb anonüümne funktsioon sümbolist `=>`, mille vasakul poolel on funktsiooni argumendid ja paremal poolel funktsiooni keha [CB15, lk 24]:

```
scala> (x: Int, y: Int) => x == y
```

2.6 Karritamine ja funktsiooni osaline rakendamine

Karritamine *currying* on protsess, mille käigus teisendatakse funktsioon, mis võtab mitu argumenti, funktsiooniks, mis võtab ühe argumendi ning tagastab uue funktsiooni, mis võtab ülejäänud argumendid, juhul kui esialgne funktsioon vajab veel argumente. Programmeerimiskeeles Haskell on kõik funktsioonid vaikimisi karritatud, mis tähendab, et kõik funktsioonid Haskellis võtavad ainult ühe argumendi [Hasb].

Scalas saab funktsioone karritada, kui jagada funktsiooni argumendid mitme argumendirühma vahel. Näiteks funktsiooni

```
def dropWhile[A](l: List[A], f: A => Boolean): List[A]
```

karritatud variant on

```
def dropWhile[A](as: List[A])(f: A => Boolean): List[A]
```

Karritatud variandi väljakutse `dropWhile(xs)(f)`, kus `xs` on suvaline list tüüpi `A` ning `f` suvaline funktsioon tüüpi `A => Boolean`, tähendab seda, et `dropWhile(xs)` tagastab funktsiooni, millele omakorda antakse argumendiks funktsioon `f`.

Nii Scalas kui Haskellis kasutatakse karritamist olukordades, kus on vaja funktsioone osaliselt rakendada. Kuid Scalas on karritamisel veel üks oluline ülesanne. Nimelt liigub karritatud funktsiooni puhul tüübiinfo automaatselt mööda argumendigruppe vasakult paremale, mis teeb Scalas, kus võrreldes Haskelliga on tüübituletus niigi väga piiratud, võimalikuks funktsiooni argumendi tüübi automaatse tuletamise. See tähendab, et karritatud `dropWhile`-funktsioonile võib argumendiks anda anonüümse funktsiooni `x => x < 3` muutuja `x` tüüpi täpsustamata. Esimese argumendigrupi põhjal on siis selge, et muutuja x tüüp on `Int`, millele funktsioon $(<)$ tõesti on defineeritud ja seda täpsustama ei pea [CB15, lk 37-38]:

```
scala>val xs: List[Int] = List(1,2,3,4,5)
xs: List[Int] = List(1,2,3,4,5)
scala>val ex1 = dropWhile(xs)(x => x < 3)
ex1: List[Int] = List(3,4)
```

Karritamata `dropWhile`-funktsiooni puhul ei saa Scala tüübituletusega ise hakkama ja nii annaks rida

```
val ex1 = dropWhile(xs, x => x < 3)
```

muutujale `x` viidates veateate *missing parameter type*.

Funktsiooni osaline rakendamine Scalas tähendab, et mitme argumendi rühmaga funktsiooni kutsutakse välja väiksema argumentide rühma arvuga, kui sellel funktsioonil on defineeritud [Docd].

2.7 Mustri sobitamine ja *case*-klassid

Mustri sobitamine ehk *pattern matching* on tegevus, mille käigus analüüsitakse etteantud avaldise struktuuri ja püütakse leida sellele vastav muster mingist mustrite hulgast. Kui sobiv muster on leitud, siis väärtustatakse sellele mustrile vastav avaldis. Kui sobivat mustrit ei leita, väljastatakse Scalas *MatchError*. Kui etteantud avaldis sobib mitme mustriga, valitakse neist esimene [CB15, lk 33].

Süntaktiliselt koosneb mustri sobitamine avaldisest, millele sobivat mustrit otsitakse, võtmesõnast `match`, millele järgneb looksulgude sees mustrite ja nendele vastavate avaldiste hulk, kusjuures igale mustrile eelneb võtmesõna `case`:

```
scala> List(3, 2, 1) match {case 3 :: _ => "algab kolmega" }
res0: String = algab kolmega
scala> List(3, 2, 1) match {case _ :: xs => xs }
res1: List[Int] = List(2, 1)
```

```
scala> List(3) match {case x :: Nil => "yheelemendiline" }
res2: Int = yheelemendiline
```

Muster võib koosneda [CB15, lk 33]:

- literaalidest (näiteks `3` ja `"neli"`),
- muutujatest (näiteks `x` ja `xs`),
- andmetüübi konstruktoritest (näiteks listi konstruktorid `x :: xs` ja `Nil`),
- eelneva kolme kombinatsioonist (näiteks `x1 :: 2 :: Nil`).

Kokkuleppeliselt kasutatakse muutujana veel alakriipsu (`_`), kui selle muutuja väärtust vastava avaldise väärtustamisel vaja ei ole [CB15, lk 33].

Nagu juba mainitud, võib mustri sobitamisel kasutada ka andmetüübi konstruktoreid. Et seda saaks teha, tuleb andmetüübi konstruktorid defineerida *case*-klasside abil. *Case*-klassid erinevad tavalistest klassidest selle poolest, et on vaikimisi mittemuudetavad ning neid võrreldakse struktuuri, mitte mäluviida järgi. Mustri sobitamisel saab *case*-klasse eraldada väiksemateks alamavaldisteks, millest see klass on moodustatud [Docc].

2.8 Algebralised andmetüübid

Sõna „algebraline“ tähendab, et andmestruktuur on moodustatud andmetüüpe kombineerides viisil, mis sarnaneb algebralisele korrutamisele ja liitmisele [Hasa].

See sarnasus pole juhuslik – andmetüüpide korrutamise ja liitmise ning numbrite korrutamise ja liitmise vahel on sügavam seos, mida põhjalikumalt uuritakse kategooriateoorias [CB15, lk 44].

Funktsionaalprogrammeerimises on algebraline andmetüüp (*algebraic data type*) vähemalt ühe konstruktoriga defineeritud andmetüüp, kus iga konstruktor võtab null või rohkem argumenti. Öeldakse, et andmestrukuur on oma konstruktorite summa või ühend ning iga konstruktor on oma argumentide korrutis, sellest ka nimetus [CB15, lk 44].

Haskellis võib korrutise ja summa defineerida järgmiselt:

```
data Pair a b = P a b
data Either a b = Left a | Right b
```

Korrutis on niisiis paar tüüpidest `a` ja `b` koos ning summa on kas tüüp `a` või `b`, aga mitte mõlemad. Haskelli `Maybe` ja `List` on näited algebralistest andmetüüpidest [Mil15]:

```
data Maybe a = Nothing | Just a
List a = Nil | Cons a (List a)
```

2.9 Kõrgemat järku funktsioonide üldistamine

Paljud algebraliste andmestruktuuridega töötamiseks mõeldud funktsioonid on üksteisele väga sarnased:

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)
}


def product(ds: List[Double]): Double = ds match {
  case Nil => 1.0
  case Cons(x, xs) => x * product(xs)
}
```

Esimene neist leiab listi elementide summa ja teine korrutise. Ainsad erinevused nende kahe funktsiooni juures on tühja listi (`Nil`) korral tagastatav väärtus ja operaator, millega kahte elementi kombineeritakse. Tuleb välja, et selliseid (ja paljusid teisi) funktsioone saab üldistada üheks kõrgemat järku funktsiooniks nimega `fold`. Funktsioon `fold` võtab lisaks listile argumendiks tühja listi korral tagastatava väärtuse ja funktsiooni, mis listi elemente kombineerib. Seda võib implementeerida kas funktsioonina `foldRight` või `foldLeft` (`foldRight` on antud koos võimaliku implementatsiooniga) [CB15, lk 37]:

```
def foldRight[A,B](as: List[A], z: B)(f: (A, B) => B): B =
as match {
  case Nil => z
  case Cons(x, xs) => f(x, foldRight(xs, z)(f))
}

def foldLeft[A,B](as: List[A], z: B)(f: (B, A) => B): B,
```

kus `z` – algväärtus; `f` – binaarne tehe, `B` – binaarse tehte `f` vastuse tüüp, `A` - andmestruktuuri elementide tüüp, `as` – list, millele funktsioone rakendatakse.

`FoldLeft` ja `FoldRight` on kõrgemat järku funktsioonid, mis töötlevad andmestruktuuri elemente mingis kindlas järjekorras ning moodustavad selle töötluse põhjal tagastusväärtuse. `FoldRight` kombineerib rekursiivselt andmestruktuuri esimese elemendi andmestruktuuri ülejäänud elementide kombineerimisel saadud tulemusega. `FoldLeft` kombineerib rekursiivselt andmestruktuuri kõigi elementide, välja arvatud viimase elemendi, kombineerimisel saadud tulemuse viimase elemendiga. `FoldRight` puhul kasutatakse algväärtust `z` rekursiivsete väljakutsetega andmestruktuuri lõppu jõudes ning `FoldLeft` puhul kombineeritakse algväärtus andmestruktuuri esimese elemendiga. Kui binaarne tehe `f` on andmestruktuuri elementide suhtes assotsiatiivne, tagastavad `foldRight` ja `FoldLeft` sama vastuse [Hasc].

Parempidist `fold`-funktsiooni võib vaadelda ka kui andmestruktuuri konstruktorite asendamist algväärtusega `z` ja funktsiooniga `f` [Hasc].

Järgnevalt on välja toodud veel mõned olulised üldistatud kõrgemat-järku funktsioonid (defineeritud listide jaoks):

```
def map[A,B](as: List[A])(f: A => B): List[B]
```

`map` rakendab etteantud listi kõigile elementidele funktsiooni `f` ning tagastab saadud listi.

```
def filter[A](as: List[A])(f: A => Boolean): List[A]
```

`filter` eemaldab listist kõik elemendid, mis ei vasta etteantud predikaadile.

```
flatMap[A,B](as: List[A])(f: A => List[B]): List[B]
```

`flatMap` erineb funktsioonist `map` selle poolest, et funktsioon `f` tagastab üksiku väärtuse asemel andmestruktuuri (antud näites listi), mille elemente kasutatakse tagastatava andmestruktuuri loomiseks [CB15, lk 42].

2.10 *for*-komprehensioon

Kuigi eelmises peatükis tutvustatud kõrgemat-järku funktsioonid nagu näiteks `map`, `filter` ja `flatMap` on programmeerijale väga kasulikud, muudab nende liigne kasutamine koodi raskesti loetavaks. Näiteks järgnevast funktsioonist oleks ilma signatuuri vaatamata üsna raske aru saada [CB15, lk 60]:

```
def map2[A,B,C](a: List[A], b: List[B])(f: (A, B) => C):  List[C
    ]=
  a flatMap (aa => b map (bb => f(aa, bb)))
```

Koodi loetavuse parandamiseks on mõnikord kõrgemat-järku funktsioonide asemel parem kasutada nendega samaväärset `for`-komprehensiooni. `For`-komprehensioon näeb välja nagu imperatiivsetest keeltest tuntud `for`-tsükkel, kuid erinevalt `for`-tsüklist tagastab see kõigi iteratsioonide tulemustest moodustatud listi. Nii näeks eelnev funktsioon `for`-komprehenisooni kasutades välja selline [CB15, lk 60]:

```
def map2[A,B,C](a: List[A], b: List[B])(f:(A, B) => C): List[C]=
  for {
    aa <- a
    bb <- b
  } yield f(aa, bb)
```

For-komprehensiooni üldkuju on `for ( s ) yield` *e*, kus *s* on järjend generaatoritest, definitsioonidest ja filtritest. Generaatori üldkuju on `val x <- e`, kus `e` on list, mille iga element järgemööda omistatakse muutuja `x` väärtuseks. Definitsiooni kuju on `val x = e`, kus muutujale `x` omistatakse kogu komprehensiooni ajaks avaldise `e` väärtus. Filtriks nimetatakse `Boolean` tüüpi avaldist `f`, mis jätab lõpptulemusest välja kõik elemendid, mis predikaadile `f` ei vasta [Ode14, lk 79-80].

3 Olulised mõisted

Selles peatükis defineeritakse tüübiklassi mõiste. Lisaks käsitletakse veel kaht olulist tüübiklassidega seotud teemat. Esiteks varjatud parameetreid ja nende kasulikkust tüübiklassidega programmeerimisel ja teiseks kõrgemat järku tüübikonstruktoreid, millega tuleb kokku puutuda teatud tüübiklasside defineerimisel.

3.1 Tüübiklass

Enamik Haskellis esinevast polümorfismist liigitub kas parameetriliseks polümorfismiks või *ad-hoc* polümorfismiks. Tüübiklassid võimaldavad Haskellis *ad-hoc* polümorfismi ühte alamliiki *overloading* struktuurselt realiseerida [Hasf].

Overloading ehk ülelaadimine on polümorfismi liik, kus funktsioonile või operaatorile võib argumendiks anda mitmest erinevast tüübist väärtusi, sest iga sellise tüübi jaoks on see funktsioon või operaator eraldi defineeritud [Hasf].[1] Tähtsaim erinevus parameetrilise polümorfismi ja ülelaadimise vahel on see, et kui parameetriliselt polümorfsed funktsioonid kasutavad kõigi tüüpide korral ühte algoritmi, siis ülelaetud funktsioonid võivad kasutada iga argumenditüübi jaoks erinevat algoritmi [Mit, lk 151].

Ülelaadimist võimaldavate funktsioonide nimed ja signatuurid antakse tüübiklassides ning neid funktsioone nimetatakse tüübiklassi mõttes polümorfseteks [Yor13]. Kuna tüübiklassis on antud ainult funktsiooni nimi ja signatuur, tuleb iga tüübi jaoks, mille peal tahetakse tüübiklassis olevaid funktsioone kasutada, need funktsioonid eraldi defineerida. Kui mingi tüübi jaoks on kõik tüübiklassis A olevad funktsioonid defineeritud, siis öeldakse, et see tüüp on tüübiklassi A isend või kuulub tüübiklassi A.

Haskelli standardmoodulis Prelude on näiteks võrdsuse operaatorit (`==`) sisaldav tüübiklass `Eq` defineeritud nii:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Seda definitsiooni võib lugeda järgmiselt: `Eq` on tüübiklass ühe parameetriga `a`. Iga tüübi jaoks, mis soovitakse teha tüübiklassi `Eq` isendiks, tuleb defineerida kaks funktsiooni, `(==)` ja `(/=)`, tüübiklassis kirjeldatud signatuuridega [Yor13].

Tüübiklassi funktsioonidele omakorda rakendub tüübiklassi kitsendus. Eelmises näites toodud funktsiooni `(==)` signatuur on selline:

```
(==) :: Eq a => a -> a -> Bool
```

[1] Haskell võimaldab tegelikult lisaks funktsioonidele ka väärtuste ülelaadimist.

Selles näites sümbolile `=>` eelnev `Eq a` on tüübiklassi kitsendus. See tähendab, et funktsioonile `(==)` võib argumendiks anda kaks väärtust suvalisest tüübist `a`, seni kuni on täidetud tingimus, et tüüp a kuulub tüübiklassi `Eq` [Yor13].

3.2 Implitsiitsed teisendused ja parameetrid

Implitsiitsed ehk varjatud teisendused (*implicit conversions*) ja parameetrid (*implicit parameters*) on Scalas kasutusel koodi kirjutamise ja lugemise lihtsustamiseks. Need võimaldava koodist välja jätta triviaalseid kohti, mis segavad programmeerijat tegelikult huvitavatele osadele keskendumast. Implitsiitsust realiseeritakse Scalas võtmesõna `implicit` abil [OSV08].

Implitsiitse teisenduse abil saab teha mingi tüübi näiliselt teise tüübi alamtüübiks, ilma et esimene tegelikult oleks teise alamtüüp. Seda kasutatakse olukordades, kus andmetüüpide muutmine pole võimalik, sest need on näiteks pärit mõnest teisest teegist [OSV08].

Kui funktsiooni parameetrigrupp implitsiitseks muuta, võib selle funktsiooni väljakutsel vastava parameetrigrupi ära jätta. Sellisel juhul püüab kompilaator puuduvatele argumentidele ise väärtused leida. Implitsiitsete parameetrite abil on näiteks mugav anda funktsioonile lisainformatsiooni mingi selle funktsiooni argumendi tüübi kohta [OSV08]. Seega võib funktsioonile implitsiitselt tüübiklassi isendeid argumendiks andes seada teistele funktsiooni argumentidele piirangu, et need peavad kuuluma vastavatesse tüübiklassidesse. Scalas kasutatakse seda ideed Haskellist tuntud tüübiklassi kitsenduste realiseerimiseks.

Näiteks olgu Scalas tüübiklass[2]:

```
trait Monoid[A] {
  def empty: A;
  def combine(x: A, y: A): A
}
```

Selles tüübiklassis on funktsioon `combineAll`, mis võtab argumendiks listi `A` tüüpi elementidest eeldusel, et `A` kuulub `Monoid`-tüübiklassi. Kompilaatori veenmiseks, et `A` tõesti kuulub `Monoid`-tüübiklassi, tuleb tõend `Monoid`-tüübiklassi kuulumisest funktsioonile argumendina kaasa anda. Selleks tõendiks on `Monoid[A]` tüüpi argument `ev`:

```
def combineAll[A](list: List[A], ev: Monoid[A]): A = {
     list.foldRight(ev.empty)(ev.combine)
}
```

Selline `combineAll` signatuur garanteerib, et `A` on monoid, kuid muudab koodi kirjutamise kohmakamaks - iga kord funktsiooni välja kutsudes tuleb kompilaatorit

[2]Tegemist on Catsi Monoid-tüübiklassi lihtsustatud definitsiooniga. Monoididest kirjutatakse lähemalt peatükis 4.2.

uuesti veenda, et `A` on monoid. Taoliste korduste vältimiseks võib funktsiooni muuta karritamise abil kahe argumendigrupiga funktsiooniks ning lisada teise grupi ette võtmesõna `implicit`:

```
def combineAll[A](list: List[A])(implicit ev: Monoid[A]): A =
    list.foldRight(ev.empty)(ev.combine)
```

Sellise signatuuriga `combineAll` funktsioonile võib argumendiks anda ainult listi – kompilaator püüab nüüd ise leida väärtuse argumendile `ev`. Et aga kompilaator puuduva väärtuse leida suudaks, tuleb ka see väärtus ise võtmesõa `implicit` abil defineerida.

Näiteks kuulugu tüüp `Int` `Monoid`-tüübiklassi:

```
implicit val intAdditionMonoid: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 0
  def combine(x: Int, y: Int): Int = x + y
}
```

Nüüd võib funktsiooni `combineAll` täisarvude listi peal välja kutsuda ilma viimase argumendita – kompilaator asendab selle implitsiitse väärtusega `intAdditionMonoid`:

```
combineAll(List(1,2,3))
// res0: Int = 6
```

3.3 Kõrgemat järku tüübikonstruktorid

Tüübikonstruktorid on Scalas vahend tüüpide loomiseks. Tüübikonstruktorid võtavad üks või rohkem tüübiargumenti ja moodustavad nende põhjal uue tüübi [WS13]. Näiteks `List` ja `Option` ei ole Scalas tüübid, vaid ühe argumendiga ehk unaarsed tüübikonstruktorid. Rakendades tüübikonstruktorit `List` tüübile `Int`, saadakse tüüp `List[Int]`.

Nii nagu iga väärtus on mingit tüüpi, on iga tüübikonstruktor mingit liiki (*kind*). Näiteks `List` on liiki `* -> *`. See tähendab, et `List` võtab mingi tüübi ja tagastab selle alusel mingi teise tüübi [WS13].

Tüübikonstruktorit, mis võtab argumendiks samuti tüübikonstruktori, nimetatakse kõrgemat järku tüübikonstruktoriks (*higher-order type constructor* ehk *higher-kinded type*) [CB15, lk 183]. Sellise tüübikonstruktori liik on `(* -> *) -> *`. See tähendab, et see tüübikonstruktor võtab mingi unaarse tüübikonstruktori ja tagastab selle alusel mingi tüübi [WS13].

Kõrgemat järku tüübikonstruktoritega tuleb enamasti kokku puutuda siis, kui abstraheerida üle tüübikonstruktorite. Näiteks teegis Cats teeb seda `Functor`-nimeline tüübiklass[3]. Sinna kuuluvad kõik tüübid, mille peal saab funktsiooni `map` kasutada:

[3]Lähemalt kirjutatakse funktoritest peatükis 4.3.

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

Ülaltoodud definitsioonis tüübiargumendi `F[_]` juures alakriips näitabki, et `F` pole mitte tüüp, vaid ühe argumendiga tüübikonstruktor [CB15, lk 188]. Kuna `Functor` võtab argumendiks tüübikonstruktori ehk üldistab üle tüübikonstruktorite, mitte tüüpide, siis öeldakse, et `Functor` on kõrgemat järku tüübikonstruktor.

4 Catsi tüübiklassid

Selles peatükis uuritakse lähemalt viit Catsi tüübiklassi. Üleliigse müra vältimiseks on siin Catsi tüübiklasside definitsioone natuke lihtsustatud. Tähelepanu tasub selles peatükis pöörata ka terminoloogiale: tüübiklasside nimed on kirjutatud `monospace` kirjastiiliga. Konkreetsesse tüübiklassi kuuluvat tüüpi nimetatakse tüübiklassi nimest tuletatud nimisõnaga. See tähendab, et näiteks tüübiklassi `Monoid` kuuluvat tüüpi nimetatakse monoidiks, tüübiklassi `Functor` kuuluvat tüüpi funktoriks jne.

4.1 Poolrühm

Öeldakse, et tüüp `A` moodustab poolrühma, kui selle tüübi jaoks leidub assotsiatiivne binaarne tehe, mis kombineerib kaks `A` tüüpi väärtust üheks `A` tüüpi väärtuseks [conh]:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}
```

Assotsiatiivsuse tõttu peab iga `A` tüüpi `x`, `y` ja `z` korral kehtima:

```
combine(x, combine(y, z)) = combine(combine(x, y), z)
```

Funktsiooni `combine` jaoks on Catsis olemas ka infiks-operaator `|+|` [conh].

Näiteks on poolrühm täisarvude hulk, kus tehteks on liitmine, samuti liitmistehtega mittetühjade listide hulk [Wikb]. Paljud programmeerijale huvipakkuvad tüübid, mis vastavad poolrühma reeglitele, on tegelikult monoidid.

4.2 Monoid

Monoid on ühikelemendiga poolrühm. Lisaks poolrühma assotsiatiivsuse reeglile peab leiduma A tüüpi element nii, et see element on vastava assotsiatiivse tehte suhtes ühikelement [cong]:

```
trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

Selles definitsioonis `empty` ongi ühikelement. See tähendab, et mistahes `A` tüüpi väärtuse `x` korral peab kehtima järgnev reegel [cong]:

```
combine(x, empty) = combine(empty, x) = x
```

Monoidid on näiteks täisarvude hulk, kus tehteks on liitmine ja ühikelemendiks 0 ning sõnede hulk, kus tehteks sõnede konkatenatsioon ja ühikelemendiks tühi sõne. Samuti listide hulk, kus tehteks on listide liitmine ja ühikelemendiks tühi list, hulkade hulk ja paljud muud andmestruktuurid koos ühikelemendi ja assotsiatiivse tehtega [Wika]. Järgnevas näites tehaksegi `String` `Monoid`-tüübiklassi isendiks:

```
val stringMonoid = new Monoid[String] {
  def combine(a1: String, a2: String) = a1 + a2
  def empty = ""
}
```

Tuleb välja, et `combine` ja `empty` sobivad ideaalselt funktsioonile `fold` argumentideks, kui enne teha eeldus, et selle signatuuris `A` ja `B` on sama tüüpi [CB15, lk 178]. Niisuguse eelduse korral näiteks funktsioon `foldRight` näeks välja selline:

```
def foldRight[A](as: List[A], z: A)(f: (A, A) => A): A
```

Tasub tähele panna, et monoidiliste tüüpide puhul polegi tegelikult oluline, kas kasutada funktsiooni `foldLeft` või `foldRight`. Tehte assotsiatiivsuse ja ühikelemendi olemasolu tõttu on tulemus mõlemal juhul sama. Seega võib listi elementide akumuleerimiseks monoidi abil kirjutada ühe üldise funktsiooni [CB15, lk 179]. Catsis on see funktsioon nimega `combineAll` ka olemas:

```
def combineAll[A](as: List[A])(implicit m: Monoid[A]): A =
  as.foldLeft(m.empty)(m.combine)
```

See funktsioon võtab listi monoidilist tüüpi elementidega ja kombineerib neid vastava monoidi assotsiatiivse tehte abil [cong]. Algväärtuseks `z` on selle monoidi ühikelement.

Veel üks huvitav funktsioon, mis implitsiitselt monoide kasutab, on tüübiklassi `Foldable` funktsioon `foldMap` [Doca]:

```
def foldMap[A, B](as: List[A])(f: A => B)(implicit B: Monoid[B])
    : B
```

Nagu nimigi vihjab, siis on `foldMap` kombinatsioon funktsioonidest `fold` ja `map`. Kõigepealt rakendatakse listi `as` igale elemendile funktsiooni `f` ja siis akumuleeritakse saadud väärtused vastava monoidi abil üheks `B` tüüpi väärtuseks. Üks võimalik kasutusviis sellele funktsioonile on olukord, kus listi peal on vaja `fold`-funktsiooni kasutada, aga selle elementide tüüp `A` ei moodusta monoidi. Seega enne tuleb elemendid teisendada tüübist `A` tüüpi `B`.

Monoidid sobivad oma omaduste tõttu hästi kasutamiseks ka paralleelarvutustes ning kuna monoidide kompositsioon on samuti monoid, siis saab mitmest lihtsamast monoidist moodustada ühe keerulisemat arvutust teostava monoidi [CB15, lk 175].

Seega töötavad programmeerijad monoidiliste tüüpidega igapäevaselt, olgu nad sellest teadlikud või mitte. Ja nii nagu kõikide abstraktsioonide puhul programmeerimises, seisneb ka monoidide kasulikkus eelkõige selles, et nad võimaldavad kirjutada geneerilist koodi, kus ainsad eeldused sisendi kohta on ainult need, mida monoid kirjeldab [CB15, lk 178].

4.3 Funktor

Scala standardteegis on paljude erinevate tüüpide jaoks olemas funktsioon `map`. Signatuuridelt on need funktsioonid teineteisele väga sarnased [CB15, lk 187]:

```
def map[A,B](as: List[A])(f: A => B): List[B]

def map[A,B](as: Option[A])(f: A => B): Option[B]
```

Ainsaks erinevuseks on tüübikonstruktorid, mille abil esimese argumendi tüüp saadakse - ühel juhul `List` ja teisel juhul `Option`. Järelikult võib tüübid, mille peal saab `map`-funktsiooni kasutada, üldistada - need on unaarsete tüübikonstruktorite abil saadud tüübid. Edasi võib sellest üldistusest lähtudes teha tüübiklassi, kuhu kuuluksid kõik sellised tüübid [CB15, lk 188]:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

Näiteks Option kuulub sellese tüübiklassi [conf]:

```
implicit val optionFunctor: Functor[Option] = new Functor[Option
    ] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa match
      {
    case None    => None
    case Some(a) => Some(f(a))
  }
}
```

Funktor peab vastama kahele reeglile [conf]:

- `fa.map(x => x) = fa`,
- `fa.map(f).map(g) = fa.map(f.andThen(g))`.

Esimene reegel ütleb, et kui rakendada funktorile identsusfunktsiooni, saadakse tulemuseks esialgse funktoriga samaväärne funktor [Lip11, lk 180]. Ühtlasi tagab see reegel, et `map` säilitab funktori struktuuri. Nii ei saa `map` näiteks listi esimest elementi eemaldada ega `Some`-väärtust `None`-väärtuseks muuta [CB15, lk 189].

Teise reegli kohaselt võib kahe järjestikuse `map`-funktsiooni kasutamise asendada ühe `map`-funktsiooniga, mille argumendiks on eelnevalt kasutatud funktsioonide kompositsioon [conf].

Sageli kasutatakse tüübikonstruktoreid selleks, et kirjeldada tüübiargumendi mingeid lisaomadusi, mida see tüüp ise ei võimalda väljendada [CB15, lk 209]. Näiteks kui funktsiooni tagastustüüp on `Option[Int]`, siis tähendab see, et funktsioon peaks tagastama `Int` tüüpi väärtuse, kuid funktsiooni sees tehtav arvutus võib ka ebaõnnestuda ja siis tagastusväärtus puudub. Lihtne näide selle kohta

on sõnastikust võtme järgi väärtuse otsimine - kui sõnastikus on see võti olemas, tagastatakse `Some(väärtus)`, vastasel juhul `None`. Sellepärast nimetavad funktsionaalprogrammeerijad tüübikonstruktoreid mõnikord ka efektideks - need näitavad, et mingi puhta väärtuse puhul tuleb arvestada efekti või lisaomadusega [CB15, lk 209]. Veel nimetatakse funktorit arvutuslikuks kontekstiks, mille sees mingit väärtust hoitakse ja kus kontekst kirjeldabki seda efekti [Lip11, lk 175].

Funktor on seega vahend, mille abil saab programmeerija töötada üksiku efektiga väärtusega, samal ajal konteksti või efekti säilitades. Selle funktsioon `map` on tööriist, millega mingit funktsiooni konteksti sees olevale puhtale väärtusele rakendada [conf].

Teisest küljest pole `map` programmeerijale midagi uudset, sest Scala standardteegis on see paljude tüüpide jaoks juba olemas. Ainuüksi selle funktsiooni pärast tüüpe funktoriteks teha polegi vaja. Samas tuleb `Functor`-tüübiklassiga kaasa veel mitu huvitavat ja kasulikku funktsiooni, mida standarteegis ei ole. Näiteks `fproduct`, `lift`, `void` ja `as` [Docb].

Catsi `map`-funktsiooni ekvivalent Haskellis on sealse `Functor`- tüübiklassi funktsioon `fmap` [Lip11, lk 112]:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Funktsioon `fmap` võtab funktsiooni tüüpi `a -> b` ja teisendab selle funktsiooniks `f a -> f b`. Sellise teisendusega saab puhtaid väärtuseid kasutavaid funktsioone mugavalt teisendada kontekstiga väärtuseid kasutavateks funktsioonideks.

Catsi `map`-funktsiooniga teistsuguse argumentide järjekorra tõttu niisugust funktsiooni teisendust teha ei saa. Küll aga on Catsi `Functor`-tüübiklassis sama eesmärgi saavutamiseks olemas funktsioon `lift` [conf]:

```
def lift[A, B](f: A => B): F[A] => F[B] = fa => map(fa)(f)
```

Nii ei pea hakkama näiteks objekti `math` funktsioone uuesti kirjutama, kui sisendi ja väljundina on tüübi `Int` asemel vaja kasutada tüüpi `Option[Int]` [CB15, lk 56]:

```
val abs1: Option[Int] => Option[Int] = Functor[Option].lift(math
    .abs)
```

Inglise keeles kasutatakse sellise funktsiooni teisenduse kohta mõistet *lifting* [CB15, lk 56].

4.4 *Applicative*

`Applicative`-tüübiklassi saab defineerida mitme erineva funktsioonide komplekti abil. Sellepärast vaadeldakse siin peatükis esmalt, kuidas on `Applicative` defineeritud puhtas funktsionaalprogrammeerimiskeeles Haskell. Seejärel uuritakse erinevaid võimalusi selle tüübiklassi defineerimiseks Scalas ning siis juba tuuakse konkreetselt Catsi vastav definitsioon.

4.4.1 *Applicative* Haskellis

Haskellis on `Applicative`-tüübiklassi definitsioon järgmine:

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Nagu definitsioonist näha, siis selleks, et tüübikonstruktori `f` saaks teha tüübiklassi `Applicative` isendiks, peab see kõigepealt kuuluma `Functor`-tüübiklassi. Seega on iga *applicative* (öeldakse ka *applicative functor*) ühtlasi ka funktor.

Definitsiooni esimene funktsioon `pure` võtab argumendiks suvalist tüüpi `a`, annab selle tüübikonstruktorile `f` tüübiargumendiks ja tagastab tüübi `f a`. Võttes aluseks peatükis 4.3 toodud võrdluse funktorist kui arvutuslikust kontekstist, võib mõelda, et `pure` võtab suvalise tüübi ja mähib selle konteksti sisse [Lip11, lk 185].

Teine funktsioon `(<*>)` võtab argumendiks funktori, mis hoiab endas funktsiooni `a -> b` ning funktori, mis hoiab endas tüüpi `a`, eraldab esimesest funktorist piltlikult öeldes funktsiooni ning rakendab seda funktorile `f a` [Lip11, lk 185]. See on justkui natuke täiustatud versioon funktsioonist `map`, sest nüüd on ka funktorile rakendatav funktsioon ise funktori sees.

4.4.2 Võimalikud definitsioonid

Haskelli `Applicative`-tüübiklassi funktsiooni `(<*>)` kohta kasutatakse Scalas põhiliselt nimesid `ap` ja `apply`. Üks võimalus `Applicative`-tüübiklassi Scalas defineerida ongi sarnaselt Haskellile funktsioonide `pure` ja `ap` abil. Kuid osutub, et tegelikult on olemas veel kaks funktsioonide komplekti, millega saaks seda tüübiklassi defineerida. Üks võimalus oleks asendada `ap` sellega sama võimsa funktsiooniga `map2` ja teine võimalus oleks `ap` asendada funktsioonidega `map` ja `product` [Lou16]:

```
pure[A](x: A): F[A]
ap[A, B](ff: F[A => B])(fa: F[A]): F[B]

pure[A](x: A): F[A]
map2[A, B, Z](fa: F[A], fb: F[B])(f: (A, B) => Z): F[Z]

pure[A](x: A): F[A]
map[A, B](fa: F[A])(f: A => B): F[B]
product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
```

Kõik need kolm funktsioonide komplekti on teineteisega samaväärsed ja sobivad `Applicative`-tüübiklassi defineerimiseks. Samaväärsus tähendab, et näiteks funktsiooni `map2` saab defineerida funktsioonide `unit` ja `ap` kaudu ning ka vastupidi - funktsiooni `ap` saab defineerida funktsioonide `map2` ja `unit` kaudu [CB15, lk 207]. Funktsioon `product` üksi nii võimas kui `ap` ja `map2` on, ei ole, mistõttu tuleb sellele juurde lisada `map` [Lou16].

Erinevates olukordades võib programmeerija eelistada kasutada erinevaid funktsioone. Samas, kuna neid funktsioone on võimalik teineteise kaudu avaldada, pole eriti oluline, millist neist kolmest komplektist kasutada `Applicative`-tüübiklassi defineerimisel - ülejäänud funktsioonid saab siis lihtsalt primitiivide kaudu avaldada.

4.4.3 *Applicative* Catsis

Catsis on `Applicative`-tüübiklassiga seotud veel üks tüübiklass, mida Haskellis ei ole – see on `Applicative`-tüübiklassi nõrgem versioon `Apply` [cone]:

```
trait Apply[F[_]] extends Functor[F] with Cartesian[F] {
    def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
}
```

`Applicative`-tüübiklassi nõrgemaks versiooniks nimetatakse seda sellepärast, et sellest puudub funktsioon `pure`. Muus osas on `Apply` ja `Applicative` identsed. `Applicative` omakorda on Catsis `Apply` alamklass [cone]:

```
trait Applicative[F[_]] extends Apply[F] {
  def pure[A](a: A): F[A]
}
```

Selles on üks uus primitiivfunktsioon `pure`. Seega on Catsis `Applicative` samuti defineeritud funktsioonide `pure` ja `ap` abil, nii nagu Haskellis.

Lisaks on Catsis `Applicative`-tüübiklassiga veel seotud tüübiklass `Cartesian` [conb]:

```
trait Cartesian[F[_]] {
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
}
```

`Cartesian` kätkeb endas teineteisest sõltumatute efektiga väärtuste kombineerimise ideed. Eriti kasulikuks osutub selle funktsioon `product` siis, kui kasutada seda koos funktsiooniga `map`. Nimelt saab nende kahe funktsiooni abil rakendada n-aarset funktsiooni n erinevale funktorile [conb]. Täpselt sama saab tegelikult teha ka `Apply`-tüübiklassi abil. Selle illustreerimiseks `Apply` ka laiendab Catsis `Cartesian`-tüübiklassi ja funktsioon `product` on seal üle defineeritud, kasutades `Apply` funktsioone [cona]:

```
override def product[A, B](fa: F[A], fb: F[B]): F[(A, B)] =
    ap(map(fa)(a => (b: B) => (a, b)))(fb)
```

Funktsiooni `product` jaoks on Catsis olemas ka infiks-sümbol `|@|`.

4.4.4 Kasutusviisid

Erinevates Catsi Applicative-tüübiklassi tutvustavates tekstides kasutadakse praktiliste näidete jaoks rohkem funktsioone `map2` ja `product` ning võrdlemisi vähe

funktsiooni `ap`. Selle põhjal võib oletada, et ka praktikas leiavad `map2` ja `product` rohkem kasutust.

Funktsiooni `map2` on erinevates näidetes kasutatud olukordades, kus mitme argumendiga puhtale funktsioonile on vaja argumendiks anda mitu konteksti sees olevat üksteisest sõltumatut väärtust.

Näiteks olgu kaks funktsiooni `getNameOption()` ja `getPayOption()`, mis teevad mingi andmebaasipäringu ja tagastavad `Option` tüüpi tulemuse. Funktsiooni `getNameOption()` tagastustüüp on `Option[String]` ja `getPayOption()` tagastustüüp on `Option[Double]`. Olgu lisaks anoüümne funktsioon `format`, mis vormistab etteantud andmed sõneks ja millel ei ole võimalikest efektidest vaja midagi teada:

```
val format = (name: String, pay: Double) => s"$name makes $pay"
```

Funktsiooniga `map2` saab nüüd päringu tulemustele funktsiooni `format` rakendada:

```
Applicative[Option].map2(getNameOption(), getPayOption())(format
    )
```

Funktsiooniga `product` saab teha sama, kuid ehk mõnevõrra elegantsemalt, kui kasutada selle infiks-sümbolit `|@|`:

```
import cats.syntax.cartesian._
(getNameOption() |@| getPayOption()).map(format)
```

Operaatori `|@|` kasutamise eeliseks on ka see, selle abil kirjutatud koodist on kergemini välja loetav, et `getNameOption()` ja `getPayOption()` on teineteisest sõltumatud. Arvutuste sõltumatus ongi just see, mida *applicative*'ite kasutamisega monaadide asemel sageli rõhutada tahetakse [Cie16].

Päringu tulemusi saaks funktsioonile `format` rakendada ka funktsiooni `ap` abil:

```
val wrappedFun: Option[Double => String] = {
  getNameOption().map(format.curried)
}

Applicative[Option].ap(wrappedFun)(getPayOption())
```

Selles näites kõigepealt kasutatakse funktsiooni `map`, mille üldine signatuur oli järgmine:

```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

Toodud näites asendub `A` tüübiga `Option[String]` ja `B` tüübiga `Double => String`:

```
def map(fa: Option[String])(f: String => (Double => String)):
  Option[Double => String]
```

Sellega rakendatakse `Option[String]` tüüpi väärtust funktsioonile `format`, kusjuures `format` on karritatud, et sellel saaks anda vähem argumente, kui see päriselt võtab. Karritatud versiooni tüüp ongi `String => (Double => String)` Sellele ühe argumendi andmisel saadakse funktsioon tüüpi `Double => String`, mille `map` paneb `Option`-konteksti sisse. Edasi juba antakse see funktsioonile `ap` argumendiks.

Kõik kolm eespool toodud lahendusviisi annavad sama lõpptulemuse, nii et see, millist neist kolmest kasutada, sõltub juba programmeerija eelistustest.

4.5 Monaad

Sarnaselt funktsioonile `map` on Scala standardteegis paljude andmetüüpide jaoks defineeritud funktsioon `flatMap`. Monaad on üldistus sellistest andmetüüpidest. Nagu eelmises peatükis, tuuakse ka siin esmalt Haskelli monaadi definitsioon ning siis võrreldakse seda Catsi vastava tüübiklassiga.

4.5.1 Monaad Haskellis

Haskellis on `Monad`-tüübiklassi defineeritud järgmiselt [Hase]:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

Kuigi Haskellist ajaloolistel põhjustel vastav tüübiklassikitsendus puudub, on oluline teada, et iga monaad on ühtlasi ka *applicative*, nii nagu iga *applicative* oli funktor. Seda saab kergesti tõestada, defineerides `Applicative`-tüübiklassi funktsioonid `Monad`-tüübiklassi funktsioonide abil.

Monaadist võib sarnaselt funktorile mõelda kui kontekstist. Tüübiklassi esimene ja kõige olulisem funktsioon `(>>=)`, võtab monaadilise väärtuse, s.t kontekstiga väärtuse ja rakendab sellele funktsiooni, mis võtab tavalise väärtuse ja tagastab monaadilise väärtuse.

Funktsioon `(>>)` on `Monad`-tüübiklassis juba vaikimisi implementeeritud [Lip11, lk 219]. See on funktsioon, mis võtab kaks monaadilist väärtust, ignoreerib esimest ja tagastab teise.

Funktsioon `return` teeb sama, mida `pure` `Applicative`-tüübiklassis.

Funktsiooni `fail` kasutamist soovitatakse üldiselt vältida [OSG08]. Haskell kasutab seda seesmiselt teatud liiki vea monaadilises kontekstis esitamiseks [Lip11, lk 229].

4.5.2 Võimalikud definitsioonid

Monad-tüübiklass peab sisaldama ühte minimaalset funktsioonide komplekti järgnevast kolmest [CB15, lk 191, 196, 198-199]:

```
unit[A](a: A): F[A]
flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
```

```
unit[A](a: A): F[A]
compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]

unit[A](x: A): F[A]
map[A, B](fa: F[A])(f: A => B): F[B]
join[A](mma: F[F[A]]): F[A]
```

Funktsioonile `flatMap` vastab Haskellis funktsioon `(>>=)` ja funktsioonile `unit` vastab Haskellis `return` või `Applicative`-tüübiklassi `pure`.

Enamasti defineeritaksegi `Monad` funktsioonide `unit` ja `flatMap` abil. Samas näiteks monaadide assotsiatiivuse reegel on arusaadavam, kui kasutada selles funktsiooni `compose`.

4.5.3 Monaad Catsis

Catsis on funktsioon `flatMap` tõstetud eraldi tüübiklassi:

```
trait FlatMap[F[_]] extends Apply[F] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
  def followedBy[A, B](fa: F[A])(fb: F[B]): F[B] = flatMap(fa)(_
      => fb)
}
```

Nagu näha, laiendab `FlatMap` `Apply`-tüübiklassi, mis oli `Applicative`-tüübiklassi nõrgem versioon. Funktsiooni `followedBy` Haskelli ekvivalent on `(>>)`. `Monad` omakorda laiendab tüübiklassi `FlatMap`:

```
trait Monad[F[_]] extends FlatMap[F] with Applicative[F] {
  ....
}
```

Tüübiklasside `FlatMap` ja `Monad` vahel on samasugune seos, nagu oli `Apply` ja `Applicative` vahel: `Monad` on `pure`-funktsiooniga `FlatMap`, nii nagu `Applicative` oli `pure`-funktsiooniga `Apply` [Yoka]. Erinevalt Haskellist funktsiooni `pure` Catsi `Monad`-tüübiklassis eraldi defineerima ei pea, sest see tuleb `Applicative`-tüübiklassist, mida `Monad` Catsis laiendab.

4.5.4 Kasutusviisid

Eelnevalt käsitletud `Functor`-tüübiklassi funktsioon `map` oli vahend puhta funktsiooni rakendamiseks kontekstiga väärtusele nii, et kontekst säiliks ja `Applicative`-tüübiklassi funktsioon `ap` oli vahend puhta funktsiooni rakendamiseks mitmele kontekstiga väärtusele nii, et kontekst säiliks.

Mida aga teha olukorras, kus kontekstiga väärtusele on vaja rakendada funktsiooni, mis võtab puhta väärtuse, aga tagastab efektiga väärtuse nii, et esialgne kontekst

säiliks? Lahenduseks ongi funktsioon `flatMap`. Kõigepealt rakendab `flatMap` algsele kontekstiga väärtusele funktsiooni ja siis selle tulemusena saadud topeltkontekst lamendatakse. Erinevates kontekstides omab selline tegevus erinevat tähendust.

Kui kontekstiks on `Option`, siis üks viis funktsiooni `flatMap` defineerimiseks on järgmine [Bja15, lk 76]:

```
def flatMap[A, B](fa: Option[A])(f: A => Option[B]): Option[B] =
      fa match {
  case None => None
  case Some(a) => f(a)
}
```

Kui argumendi `fa` väärtuseks on `None`, siis tagastatakse samuti `None`, sest funktsioonile `f` ei ole võimalik midagi argumendiks anda. Kui `fa` väärtuseks on `Some(a)`, siis `a` antakse funktsioonile `f` argumendiks. See omakorda tagastab `Option[B]` tüüpi väärtuse, mis ühtlasi on kogu funktsiooni tagastusväärtuseks.

Üldiselt, kui konstrueerida `Option`-tüüpi väärtuste peal `flatMap`-funktsioonide ahel, siis see ahel tagastab lõppväärtuse `None` niipea, kui mõni funktsioon, mida ahelas rakendatakse, tagastab `None`. Seega on funktsioonil `flatMap` omamoodi vookontrolli roll - edasised tegevused selles ahelas sõltuvad sellest, milline oli eelmise tegevuse tulemus.

Näiteks olgu kolm sõnastikku, millest võtme järgi väärtuse otsimisel saadakse `Option`-tüüpi tulemus, sest võti võib, aga ei pruugi vastavas sõnastikus olemas olla:

```
val idsByName: Map[String,Int]
val depts: Map[Int,String]
val salaries: Map[Int,Double]
```

Eesmärk on kirjutada programm, mis töötaja nime järgi leiaks, mis osakonnas see töötaja töötab ja kui palju ta palka saab. Selleks on vaja sooritada kolm operatsiooni: leida nime järgi töötaja id ning leida töötaja id järgi tema osakond ja töötasu, kusjuures kahe viimase operatsiooni tulemus sõltub esimese tulemusest. Kui vastava nimega töötajat ei leidu, siis pole ka millegi järgi osakonda ja töötasu leida. Funktsiooni `flatMap` kasutades näeks programm välja selline:

```
val result: Option[String] = idsByName.get("Bob").flatMap { id =
    > Applicative[Option].map2(depts.get(id), salaries.get(id))((
    dept, salary) => s"Bob in $dept makes $salary per year"
}
```

Kui selles näiteks oleks funktsiooni `flatMap` asemel kasutatud funktsiooni `map`, siis oleks result olnud tüüpi `Option[Option[String]]` ehk kontekst poleks säilinud.

Üks huvitav funktsioon, mis Catsis monaade implitsiitselt kasutab, on tüübiklassi `Foldable` funktsiooni `foldM` [conc]:

```
def foldM[G[_], A, B](fa: F[A], z: B)(f: (B, A) => G[B])(
    implicit G: Monad[G]): G[B]
```

Erinevalt funktsioonist `fold` kasutab vasakassotsiatiivne `foldM` andmestruktuuri voltimiseks binaarset funktsiooni, mis tagastab monaadilist tüüpi väärtuse. Ka funktsiooni `foldM` tagastusväärtus on seetõttu monaadilist tüüpi. Seda sobib kasutada näiteks olukorras, kus andmestruktuuri voltimisel on vaja kontrollida, et selle elemendid täidaksid mingit lisatingimust, mis lõpptulemust omakorda mõjutab [Lip11, lk 264]. Näiteks olgu vaja leida listi elementide summa tingimusel, et kui listis on leidub arvust 9 suurem element, siis kogu arvutus ebaõnnestub [Yokb]:

```
def binSmalls(acc: Int, x: Int): Option[Int] = if (x > 9) None
    else Some(acc + x)

Foldable[List].foldM(List(2, 8, 3, 1), 0)(binSmalls)
//Some(14)

Foldable[List].foldM(List(2, 11, 3, 1), 0)(binSmalls)
//None
```

5 Õppematerjal

Tartu Ülikooli Scala kursuse üks eesmärk on tutvustada võimalusi mitme-paradigma keeles programmeerimiseks, sealhulgas erinevaid funktsionaalsete keelte abstraktsioone. Kursuse läbinud tudeng võiks omada varasemast mitmekülgsemaid teadmisi probleemi modelleerimisest ja programmi disanimisest. Samuti võiks tal kursuse lõpuks olla kogemus mitte-muudetavate andmestruktuuridega töötamisega ja puhaste funktsionaalsete abstraktsioonide kasutamisega.

Loodud harjutused tutvustavad teegi Cats pakutavaid võimalusi Scalas funktsionaalsete abstraktsioonidega programmeerimiseks. Harjutuste soovitud õpiväljund on see, et need praktikumis läbi lahendanud tudeng oskaks kõige lihtsamate Catsis eeldefineeritud tüübiklasside abil ülesandeid lahendada ja mõistaks tüübiklasside eeliseid objekt-orienteeritud lähenemise ees.

5.1 Harjutusülesanded

Harjutusülesanded on koostatud pigem lihtsamad kui raskemad, sest on mõeldud lahendamiseks inimesele, kellel varasem kogemus tüübiklassidega programmeerimisega puudub. Ülesanded on grupeeritud tüübiklasside kaupa ning nende vahele on kommentaaridena lisatud võrdlemisi palju selgitavat teksti nii ülesande püstituse kui ka vastava tüübiklassi kohta üldiselt. Selle eesmärk on, et ülesandeid oskaks iseseisvalt lahendada ka inimene, kes käesolevat tööd lugenud ei ole, kuid kellel on natuke kogemust Scala ja funktsionaalprogrammeerimisega.

Järgnevalt tuuakse lühike ülevaade koostatud ülesannetest tüübiklasside kaupa. Ülesannete lähtekood on leitav lisast 1.

Monoid Esimeses ülesandes tuuakse näide `Monoid`-tüübiklassi isendist ning tutvustatakse funktsiooni `combine`. Lahendajal palutakse erinevate monoidiliste tüüpide peal seda funktsiooni katsetada.

Teises ülesandes tuleb lahendajal defineerida `Monoid`-tüübiklassi isend `Boolean`-tüübi jaoks. Selle eesmärk on harjutada tüübiklassi isendite defineerimist. Seejärel tuuakse näide kolmest väga sarnasest funktsioonist, mille asemel palutakse lahendajal kirjutada üks üldine funktsioon `combineAll`, mis kasutaks ära monoidide omadusi. Selle eesmärk on, et lahendaja õpiks nägema koodis korduvaid mustreid ja püüaks neid üldistada. Samuti näitab see ülesanne, kuidas tulevad tüübiklassid selliste üldiste funktsioonide kirjutamisel kasuks. Viimaks kasutatakse eelnevat kahte funktsiooni `forAll` defineerimiseks, mis kontrollib monoidi abil, kas listis on kõik tõeväärtused tõesed.

Kolmas ülesanne on mõeldud implitsiitselt monoide kasutava funktsiooniga `foldMap` tutvumiseks, mille abil need ülesanded lahendada tuleks.

Neljandas ülesandes on tulemuseks vaja saada klassi `Order` isend, mis järjestaks arvupaarid summa alusel ja summa sees leksikograafiliselt. Selleks tuleb kasutada tüübi `Order` jaoks defineeritud monoidi `whenEqualMonoid`.

Funktor Esimeses ülesandes tuleb defineerida `Funcor`-tüübiklassi isend enda loodud tüübi `Box` jaoks. Eesmärk on ja illustreerida väidet, et funktor on justkui kontekst mingi väärtuse ümber. Seejärel tuleb implementeerida funktsioon, mis teisendaks funktsiooni tüüpi `A => B` funktsiooniks tüüpi `F[A] => F[B]`. Eesmärk on näidata, kuidas pelgalt primitiivkombinaatorite abil saab defineerida uusi funktsioone. Viimaks tuleb lahendajal enda tehtud funktor komponeerida listi funktoriga. Selle eesmärk on näidata, et funktorite kompositsioon on samuti funktor.

Teise ülesande eesmärk on illustreerida väidet, et funktor on vahend töötamaks üksiku kõrvalefektiga. Selleks tuleb lahendajal lõpetada programm, mis küsib kasutajalt sisendi, töötleb seda ja tagastab mingi vastuse. Selles näites kasutajalt saadud sisend on efektiga ehk `Option` tüüpi, sest ei pruugi olla korrektne.

Applicative Lahendamiseks on üks pikem ülesanne, mille eesmärk on ilmestada väidet, et *applicative*'st võib mõelda kui vahendist, mis võimaldab töötada mitme sõltumatu konteksti sees oleva väärtusega korraga. Ülesande lahendaja peab kirjutama programmi, mis küsib kasutajalt kaks sisendit (saadakse kaks efektiga väärtust), töötleb neid mingi puhta funktsiooniga ja tagastab vastavalt selle tulemusele vastuse. Lahendaja peab selle idee realiseerima mitmel erineval viisil: kõigepealt kasutades funktsiooni textttap, seejäärel funktsiooniga `map2` ning viimaks `Cartesian`-tüübiklassi funktsiooni `product` abil.

Monaad Lahendamiseks on samuti üks pikem ülesanne. Sellega soovib autor näidata, kuidas `flatMap` ahela abil saab moodustada tegevuste jada, kus iga järgnev tegevus sõltub eelmise tegevuse tulemusest. Ülesandeks on kirjutada programm, mis simuleerib lindude maandumist köielkõndijal tasakaaluteiba paremale ja vasakule äärele. Programm peab väljastama, milline on lindude seis kummalgi poolel pärast teatud arvu maandumisi ja õhkutõusmisi. Arvestada tuleb aga sellega, et kui parema ja vasaku poole lindude arv erineb liiga palju, siis köielkõndija läheb tasakaalust välja ja kukub alla, mistõttu lõpptulemus puudub. Seega iga uue maandumise korral tuleb arvestada, milline oli lindude seis eelmise maandumise järel.

5.2 Näiteprogramm

Näiteprogrammi eesmärk on loengus teegi esmane tutvustamine ja selle vastu huvi tekitamine, lahendades mingit loengu kuulajatele tuttavat probleemi Catsi

abil funktsionaalsel viisil. Konkreetseks näiteks valis töö autor ankeedi andmete valideerimise Catsi andmetüübi `Validated` ja tüübiklassi `Applicative` abi.

`Applicative`-tüübiklassist võib lähemalt lugeda peatükist 4.2. `Validated` on andmetüübile `Either` sarnane Catsi andmetüüp veaohtlike tegevuste tegemiseks funktsionaalsel viisil. Kui `Either` lõpetab veaohtlike tegevuste ahelas esimese ebaõnnestunud tegevuse korral töö ja tagastab vastava veateate, siis `Validated` töötleb läbi kogu ahela, kogub kõik ette tulnud veateated kokku ja tagastab näiteks listi nendest [cond]. Sellepärast sobib `Validated` hästi kasutamiseks olukordades, kus on vaja teada, millised tegevused täpsemalt ahelas ebaõnnestusid ja mis oli ebaõnnestumise põhjuseks. Hea näide selle kohta ongi ankeet, kus küsitakse näiteks kasutaja isikuandmeid ning kus võib vaja olla mitu erinevat veateadet kasutajale väljastada, juhul kui mitme välja sisendid ei vasta reeglitele.

`Validated`-andmetüübi jaoks on olemas kaks konkreetset implementatsiooni - `Valid` ja `Invalid` [cond]:

```
sealed abstract class Validated[+E, +A]

object Validated {
  final case class Valid[+A](a: A) extends Validated[Nothing, A]
  final case class Invalid[+E](e: E) extends Validated[E,
      Nothing]
}
```

Funktsioon, mis selle abil valideerib kasutaja sisestatud nime, võiks välja näha näiteks järgmine:

```
def validName(name: String): Validation[String, String] =
  if (name != "") Valid(name)
  else Invalid("Name cannot be empty")
```

Osutub, et `Validated` kuulub ka `Applicative`-tüübiklassi ning et Catsis on vastav tüübiklassi isend implitsiitselt juba defineeritud [cond]. Seega saab näiteks mingi objekti loomiseks kasutada `Applicative`-tüübiklassi funktsiooni `map3`:

```
case class WebForm(name: String, age: Int, phoneNumber: String)

import cats.data.{ NonEmptyList => NEL }
def validName(name: String): Validation[NEL[String], String]
def validAge(age: String): Validation[NEL[String], Int]
def validPhone(number: String): Validation[NEL[String], String]

def validWebForm(name: String, age: String, phone: String):
    Validation[NEL[String], WebForm] = map3(validName(name),
    validAge(age), validPhone(phone))(WebForm(_,_,_))
```

Kui kõik funktsioonile `validWebForm` ette antud argumendid vastavad reeglitele, luuaksegi nende abil objekt `WebForm`. Vastasel juhul tagastatakse mittetühi list valideerimisel tekkinud veateadetega.

Loengunäide on idee poolest väga sarnane ülaltoodud näitele. Lähtekood on leitav lisast 2.

5.3 Tagasiside

Harjutusülesandeid tutvustati programmeerimiskeelte seminaris programmeerimise õpetamisega seotud inimestele. Eesmärk oli välja selgitada, kuivõrd ilmestavad koostatud ülesanded tüübiklasside eeliseid objektorienteeritud lähenemise ees. Samuti soovis töö autor tagasisidet ülesannete raskusastme kohta.

Tagasiside raskusastme kohta oli, et teema nõuab süvenemist ja enne lahendamist oleks mõistlik töö kirjalikku osa lugeda. Sellega on töö autor arvestanud – enne vastava praktikumi toimumist on kavas loengus tüübiklasside teemat tutvustada. Töö raames kirjutati ka näiteprogramm, millega saab loengumaterjali illustreerida.

Veel tehti ettepanek mõiste *implitsiitne* mingi muu, vähem võõra sõnaga asendada. Leiti, et antud kontekstis sobiks selle asemele kõige paremini sõna *varjatud*. Töö autor nõustus, et ka sõna *varjatud* võiks kasutada ja lisas peatükki 3.2 vastava täienduse, kuid otsustas siiski oma töös põhiliselt mõistet implitsiitne kasutada. Seda põhjusel, et sõnal *varjatud* on liiga lai tähendus ja teksti sees kasutades ei seostu see nii hästi võtmesõnaga `implicit`.

Ülesanded lahendas läbi üks programmeerimikeelte õpetamise kogemusega magistrant. Töö autor tegi lahendamise jälgimisel järgmised tähelepanekud:

- viimases monoidide ülesandes ei teadnud lahendaja süntaksit, millega Catsis juba olemasolevat paare võrdlevat `Order`-tüübiklassi isendit kätte saada;
- osa `Applicative`-tüübiklassi ülesannete juures olevaid selgitusi osutusid liiga üldiseks ja ei aidanud eriti ülesande lahendamisele kaasa;
- Haskellile sarnase punktideta süntaksi kasutamisel mitme argumendiga või karritatud funktsiooni peal tekkisid kompileerimisaegsed vead, mida lahendaja parandada ei osanud;
- polnud piisavalt selgelt välja toodud, et Catsis olemasolevate tüübiklassi isendite jaoks on kaks viisi, kuidas tüübiklassi funktsioone nende peal kasutada – isendi meetodina ja otse tüübiklassist.

Esimese kahe tähelepaneku põhjal tegi autor harjutusülesannetes muudatusi – selgitas mõningaid ülesandeid põhjalikumalt ja ning lisas kommentaaridena juurde nende funktsioonide signatuurid, mille kasutamist lahendustes nõuti. Samuti täiendas autor viimast monoidide ülesannet – lisas juurde muutuja, millele on omistatud Catsis juba olemasolev `Order`-tüübiklassi isend.

Punktideta süntaksi kasutamine Scalas ei ole kohustuslik ning selle tutvustamine jääb praktikumijuhendate ülesandeks. Siiski soovitatakse seda kasutada ainult meetodite puhul, mis võtavad ühe argumendi [Docf].

Viimases punktis toodule tuleks enne vastava praktikumi toimumist loengus kindlasti tähelepanu juhtida. Samuti tuleks loengus tutvustada süntaksit, millega otse tüübiklassist funktsioone kätte saab, sest kuigi testlahendaja mõtles selle oma kogemustele tuginedes välja, ei saa samasugust kogemust aine kuulajatelt eeldada.

6 Kokkuvõte

Käesoleva bakalaureusetöö eesmärk oli uurida teeki Cats, mis implementeerib programmeerimiskeele Scala jaoks erinevaid tüübiklasse. Lähemalt seadis töö autor eesmärgiks uurida viit põhilist Catsi tüübiklassi, et teada saada, kuidas täpsemalt on tüübiklassid Catsis realiseeritud ning mida kasulikku ja huvitavat nendega Scalas üldse teha saab. Selle uurimuse tulemusena valmis sissejuhatuslik kirjalik materjal tüübiklassidega programmeerimisest Scalas, mida täiendavad harjutusülesanded nende tüübiklasside kohta ja näiteprogramm.

Selgus, et Catsi tüübiklasside süsteem on palju keerulisem, kui näiteks puhtas funktsionaalses keeles Haskell. Seda põhiliselt seetõttu, et mõned Haskellist tuttavad tüübiklassid on Scalas jagatud mitmeks erinevaks tüübiklassiks. Ilmselt on sellel ka omad eelised, mille uurimine käesolevas töös aga eesmärgiks ei olnud.

Haskellist tuttavad tüübiklassipiirangud on Catsis realiseeritud implitsiitsete parameetritena. Catsis on mitu huvitavat implitsiitselt tüübiklasse kasutavat funktsiooni ka juba olemas. Näiteks implitsiitseid monoide kasutab funktsioon `foldMap` ja monaade `foldM`.

Lisaks leiab Catsi tüübiklassidest mitmeid kasulikke primitiivide kombineerimisel saadud funktsioone. Näiteks `Monoid`-tüübiklassi `combineAll` ja `Functor`-tüübiklassi `lift`. Samuti selgus, et tüübiklassidel on Scalas oluline roll kõrvalefektide käsitlemisel funktsionaalsel viisil.

Tüübiklasside `Functor` ja `Monad` vajalikkust Scala programmeerijale kahandas natuke asjaolu, et Scalas juba on paljude andmetüüpide jaoks funktsioonid `map` ja `flatMap` olemas. Samas on nendes tüübiklassides veel mitmeid huvitavaid funktsioone.

Tüübiklassile `Semigroup` oli praktilist kasutust keeruline leida, sest enamik sellesse tüübiklassi kuuluvaid tüüpe, mis programmeerijale huvi võiksid pakkuda, on tegelikult monoidid. Seega võiks töös vaadeldud Catsi tüübiklassidest Scala programmeerijale kõige rohkem praktilist kasu olla tüübiklassidest `Monoid` ja `Applicative`.

Lisaks koostas töö autor tüübiklassidega programmeerimise praktiseerimiseks harjutusülesanded. Ülesanded on raskusastme poolest võrdlemisi lihtsad, sest nende lahendajalt ei eeldata erilist funktsionaalprogrammeerimise kogemust. Töö käigus valmis ka Catsi tutvustamiseks väike näidisprogramm. Nii harjutusülesandeid kui ka näidisprogrammi on tulevikus plaanis kasutada loengu- ja praktikumimaterjali koostamisel.

Viited

[Bja15] Rúnar Óli Bjarnason. A companion booklet to "functional programming in scala". http://blog.higher-order.com/assets/fpiscompanion.pdf, March 2015.

[CB15] Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala*. Manning Publications Co., 2015.

[Cie16] Krzysztof Ciesielski. The underrated applicative functor. https://softwaremill.com/applicative-functor/, April 2016.

[cona] Cats contributors. Apply. GitHub. https://github.com/typelevel/cats/blob/master/core/src/main/scala/cats/Apply.scala.

[conb] Cats contributors. Cartesian. GitHub. https://github.com/typelevel/cats/blob/master/core/src/main/scala/cats/Cartesian.scala.

[conc] Cats contributors. foldm. http://typelevel.org/cats/api/cats/Foldable.html#foldM[G[_],A,B](fa:F[A],z:B)(f:(B,A)=>G[B])(implicitG:cats.Monad[G]):G[B].

[cond] Cats contributors. Validated. http://typelevel.org/cats/datatypes/validated.html.

[cone] Typelevel contributors. Applicative. http://typelevel.org/cats/typeclasses/applicative.html.

[conf] Typelevel contributors. Functor. http://typelevel.org/cats/typeclasses/functor.html.

[cong] Typelevel contributors. Monoid. http://typelevel.org/cats/typeclasses/monoid.html.

[conh] Typelevel contributors. Semigroup. http://typelevel.org/cats/typeclasses/semigroup.html.

[Doca] Cats Documentation. Foldmap. http://typelevel.org/cats/api/cats/Foldable.html#foldMap[A,B](fa:F[A])(f:A=>B)(implicitB:cats.Monoid[B]):B.

[Docb] Cats Documentation. Functor. http://typelevel.org/cats/api/cats/Functor.html.

[Docc] Scala Documentation. Case classes. http://docs.scala-lang.org/tutorials/tour/case-classes.html.

[Docd] Scala Documentation. Currying. http://docs.scala-lang.org/tutorials/tour/currying.

[Doce] Scala Documentation. Higher-order functions. http://docs.scala-lang.org/tutorials/tour/higher-order-functions.

[Docf] Scala Documentation. Style guide. method invocation. http://docs.scala-lang.org/style/method-invocation.html.

[Hasa] HaskellWiki. Algebraic data type. https://wiki.haskell.org/Algebraic_data_type.

[Hasb] HaskellWiki. Currying. https://wiki.haskell.org/Currying.

[Hasc] HaskellWiki. Fold. https://wiki.haskell.org/Fold.

[Hasd] HaskellWiki. Functional programming. https://wiki.haskell.org/Functional_programming.

[Hase] HaskellWiki. Monad class. https://wiki.haskell.org/Monad#Monad_class.

[Hasf] HaskellWiki. Polymorphism. https://wiki.haskell.org/Polymorphism.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* April 2011. Ebook.

[Lou16] Sinisa Louc. Functors and applicatives. https://hackernoon.com/functors-and-applicatives-b9af535b1440, July 2016.

[Mil15] Bartosz Milewski. Simple algebraic data types. https://bartoszmilewski.com/2015/01/13/simple-algebraic-data-types/, January 2015.

[Mit] John Mitchell. Concepts in programming languages. https://cseweb.ucsd.edu/~dstefan/cse130-winter17/book/ch07.pdf. Revised chapter.

[Ode14] Martin Odersky. *Scala By Example.* Programming Methods Laboratory EPFL Switerland, http://www.scala-lang.org/docu/files/ScalaByExample.pdf, 2014.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*, chapter 14. Monads. O'Reilly Media, 2008. http://book.realworldhaskell.org/read/monads.html#monads.class.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, First Edition*, chapter 21. Implicit Conversions and Parameters. http://www.artima.com/pins1ed/implicit-conversions-and-parameters.html, December 2008.

[Wika] Wikipedia. Examples. https://en.wikipedia.org/wiki/Monoid#Examples. Examples of monoids.

[Wikb] Wikipedia. Examples of semigroups. https://en.wikipedia.org/wiki/Semigroup#Examples_of_semigroups.

[WS13] Jed Wesley-Smith. Scala: Types of a higher kind. https://www.atlassian.com/blog/archives/scala-types-of-a-higher-kind, September 2013.

[Yoka] Eugene Yokota. Monad. http://eed3si9n.com/herding-cats/Monad.html.

[Yokb] Eugene Yokota. Some useful monadic functions. http://eed3si9n.com/herding-cats/monadic-functions.html.

[Yor13] Brent Yorgey. Type classes. https://www.schoolofhaskell.com/school/starting-with-haskell/introduction-to-haskell/5-type-classes, November 2013.

Kõiki veebilehti vaadati viimati 11.05.2017.

Lisad

1 Harjutusülesanded

Harjutusülesannete lähtekood on saadaval aadressil `https://github.com/helehh/cats-excercises`. Ülesanded on failides *MonoidEx.scala*, *FunctorEx.scala* ja *MonadEx.scala*.

2 Näidisprogrammi kood

Näidisprogrammi lähtekood on samuti saadaval aadressil `https://github.com/helehh/cats-excercises`. Programm on failis *ValidatedEx.scala*.

3 Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Hele-Andra Kuulmets,

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

   Tüübiklassidega funktsionaalprogrammeerimine Scalas,

   mille juhendaja on Vesal Vojdani,

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

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

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 11.05.2017