TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Karoliine Holter

Funktsionaalprogrammeerimise õpetamine Idrises

Bakalaureusetöö (9 EAP)

Juhendajad: Kalmer Apinis, PhD
Vesal Vojdani, PhD

Tartu 2021

Funktsionaalprogrammeerimise õpetamine Idrises

Lühikokkuvõte:

Viimase kümne aasta jooksul on funktsionaalprogrammeerimine palju arenenud ja edasi liikunud. Bakalaureusetöö eesmärk on uurida, kas sõltuvate tüüpidega keel Idris on sobilik keel bakalaureuseastmes klassikaliste ja tänapäevaste funktsionaalprogrammeerimise teemade õpetamiseks. Selleks kohandati Haskelli baasil eelkõige klassikalist funktsionaalprogrammeerimist õpetava kursuse „Programmeerimiskeeled“ praktikumiülesanded Idrisesse ning uuriti, milliseid uusi teemasid saab Idrise kasutusele võtmisega kursuse kavva lisaks võtta. Töös tutvustatakse Haskelli ja Idrise põhilisi erinevusi „Programmeerimiskeelte“ aines käsitletud teemade ulatuses ning sõltuvate tüüpidega programmeerimist Idrises. Töö põhjal jõuti järeldusele, et Idris on sobilik keel bakalaureuseastmes funktsionaalprogrammeerimise õpetamiseks.

Võtmesõnad: funktsionaalprogrammeerimine, Haskell, Idris, sõltuvad tüübid, tüübisüsteemid, verifitseerimine

CERCS: P175 Informaatika, süsteemiteooria

Teaching functional programming in Idris

Abstract:

Over the last ten years, functional programming has developed and moved forward. The Bachelor’s work aims to examine whether the dependently typed language Idris is a suitable language for teaching classical and modern functional programming subjects in a bachelor’s degree. To this end, the practical tasks of the course that were based on Haskell and mainly covered the traditional functional programming topics were adapted to Idris. It was examined which new topics could be added to the curriculum by introducing Idris. The work presents the fundamental differences between Haskell and Idris in the scope of the subjects covered by the "Programming Languages" course and introduces programming with dependent types in Idris. Based on the work, it was concluded that Idris is a suitable language for teaching functional programming at a bachelor’s level.

Keywords: dependent types, functional programming, Haskell, Idris, type systems, verification

CERCS: P175 Informatics, systems theory

Sisukord

1 Sissejuhatus **4**

2 Funktsionaalprogrammeerimine **6**

2.1 Haskell ja klassikaline funktsionaalprogrammeerimine 7

2.2 Idris ja tüübipõhine arendus 8

3 Klassikaliste teemade ülekandmine **9**

3.1 Süntaksi erinevused 10

3.2 Keelekonstruktsioonide erinevused 12

3.2.1 Sõned 12

3.2.2 Valvurid 13

3.2.3 Laiskusega seotud probleemid 15

3.2.4 Erindid 16

3.2.5 Tüübiklassid 16

3.3 Mõistlikult kohandamatud ülesanded 18

3.4 Klassikaliste teemade käsitlemine Idrises 19

4 Sõltuvate tüüpidega programmeerimine ja verifitseerimine **20**

4.1 Tüüpide arvutamine ja sõltuvate tüüpidega funktsioonid 20

4.2 Sõltuvad tüübid 24

4.2.1 Vektorid 24

4.2.2 Kujulised puud 25

4.3 Interaktiivne programmeerimine 27

4.3.1 Funktsiooni definitsiooni skeleti andmine 28

4.3.2 Juhtudeks jagamine 28

4.3.3 Augud ja otsimine 29

4.4 Valemite tüüpidena väljendamine 30

4.5 Uute teemade käsitlemine Idrises 34

5 Kokkuvõte **35**

Viidatud kirjandus **36**

Lisad **38**

I Litsents 38

1. Sissejuhatus

Funktsionaalprogrammeerimise kursus, mida Tartu Ülikoolis praegu loetakse, on üles ehitatud programmeerimiskeelte Haskell ja Scala õpetamisele ning kannab pealkirja „Programmeerimiskeeled“ [1]. Käesoleva kursuse programmeerimiskeelte baasil ei saa aga õpetada tänapäevaseid tüübisüsteemipõhiseid teemasid. Näiteks ei saa Haskellis otseselt kirjutada tüübi taseme funktsioone ja sõltuvate tüüpidega funktsioone. Haskelli laienduste kaudu saab neid teemasid kaudselt käsitleda, aga bakalaureuse taseme jaoks on vaja otsest käsitlust ning lihtsamat kirjapilti[1]. Seega otsustati analüüsida, kas kursust saab anda programmeerimiskeeles Idris. Analüüsimisel peab arvestama, et Haskellis saab väga hästi õpetada klassikalist funktsionaalset programmeerimist, mille käsitlemine ei tohi muudatuste tõttu kannatada[1].

Töö eesmärk on analüüsida, kas sõltuvate tüüpidega keel Idris on sobilik keel bakalaureuseastmes klassikalise ja tänapäevase funktsionaalprogrammeerimise õpetamiseks. Selleks sõnastati uurimisküsimus: Kuidas mõjutab Haskelli asendamine Idrisega klassikaliste funktsionaalprogrammeerimise teemade käsitlemist ning milliste tänapäevaste teemade õpetamist asendamine võimaldab?

Uurimisküsimuse esimesele osale vastamiseks analüüsiti keeltevahelisi erinevusi ning kontrolliti, kas praeguse kursuse praktikumide materjalid saab uuele keelele kohandada. Töö annab ülevaate Idrise ja Haskelli sarnasustest ja erinevustest aines „Programmeerimiskeeled“ käsitletud teemade ulatuses. Uurimisküsimuse teisele osale vastamiseks uuriti, milliste bakalaureuse tasemele kohaste uute teemade käsitlemist sõltuvad tüübid võimaldavad. Töös kirjeldatakse sõltuvate tüüpidega programmeerimise eeliseid, mis päädib funktsioonide korrektsuse tõestamisega.

Bakalaureusetöö sisuline osa on jaotatud kolmeks peatükiks numbritega kaks kuni neli. Teises peatükis antakse ülevaade funktsionaalprogrammeerimise mõistest, asjakohastest funktsionaalsetest programmeerimiskeeltest ning nende arengusuundadest. Kolmandas peatükis kirjeldatakse ja analüüsitakse olemasolevate materjalide kohandamise käigus välja tulnud Haskelli ja Idrise vahelisi sarnasusi ja erinevusi ning vastatakse

[1] Kalmer Apinis e-kirjas (05.10.2020)

uurimisküsimuse esimesele osale. Neljandas peatükis kirjeldatakse Idrise omadusi ja nendega kaasnevaid võimalusi ning antakse ülevaade, millised on võimalikud lähenemised verifitseerimise teema sissejuhatamiseks. Neljanda peatüki lõpus vastatakse uurimisküsimuse teisele osale.

2. Funktsionaalprogrammeerimine

Selles peatükis antakse esmalt ülevaade funktsionaalprogrammeerimisest ja selle üldistest mõistetest. Seejärel on toodud teemakohaste keelte lühikirjeldused ja eesmärgid ning motivatsioon, miks on mõistlik liikuda edasi ning proovida õpetada funktsionaalprogrammeerimist uue keele baasil.

Funktsionaalprogrammeerimist ja funktsionaalset programmeerimiskeelt defineeritakse erinevalt. Puhast funktsionaalset programmeerimiskeelt defineeritakse kui keelt, milles programmeeritakse kasutades matemaatilisi funktsioone [3]. Funktsionaalprogrammeerimist defineeritakse üldiselt aga kui paradigmat või programmeerimisstiili, kus keskendutakse lahenduse või tõe kirjeldamisele [1, 2]. See erineb imperatiivsest paradigmast, kus keskendutakse sellele, mis operatsioone teha [1]. Funktsionaalprogrammeerimist kui stiili on kõige parem õppida puhta funktsionaalse keele baasil, sest puhtas funktsionaalses keeles teisiti programmeerida ei saagi [4].

Puhtas funktsionaalses keeles arvutavad funktsioonid väärtuseid sõltuvalt ainult nende sisenditest [1]. See tähendab, et puhtas funktsionaalses keeles ei ole globaalseid muutujaid või andmestruktuure, mis võiksid programmi arvutuskäiku mõjutada või mida mitu erinevat funktsiooni võiksid muuta [1, 3]. Seega on funktsioonid teineteisest sõltumatud, mis võimaldab neid kergesti taaskasutada, programme modulaarselt üles ehitada kui ka nende korrektsust tõestada [2].

Klassikalisteks funktsionaalprogrammeerimise teemadeks nimetatakse selles töös teemasid, mis tulenevad otse LISPi õpetamisest ehk tüüpimata funktsionaalprogrammeerimine. Näiteks listide, paaride ja puudega kõrgemat järku funktsioonid. Tänapäevaste funktsionaalprogrammeerimise teemade all peetakse silmas sõltuvaid tüüpe, korrektsuse tõestamist ning mittemonaadilist programmeerimist, näiteks aplikatiivsed ja ühikuga funktorid või lineaarsusel põhinev sisend-väljund. Klassikaliste ja tänapäevaste teemade üleminekule jäävad tüübiklassid ja monaadid.

2.1. Haskell ja klassikaline funktsionaalprogrammeerimine

Haskell on tugevalt ning staatiliselt tüübitud, laisk, puhas funktsionaalne keel [1]. Järgnev Haskelli arengut ja eesmärke käsitlev lõik tugineb P. Hudak jt. kirjutatud artiklile „A History of Haskell: Being Lazy With Class“ [7]. Haskell on disainitud ja teostatud Haskelli komitee poolt mitmete inimeste koostöös. Haskelli esimene versioon avaldati 1990. aastal ning 1998. aastal välja antud „Haskell 98“ sai aluseks selle esimesele stabiilsele versioonile. Haskelli üks põhilisi omadusi on laiskus, mis on kahtlemata üks põhilistest teemadest, mis Haskelli disaini panustanud inimesi ühendas. Üks välja kirjutatud eesmärkidest oli, et keel peaks olema sobiv nii õpetamiseks, teadustööks kui ka rakenduste loomiseks, sealhulgas suurte süsteemide ehitamiseks [7].

Praegu meid kõige enam huvitav osa eelmainitud eesmärgist, „peaks olema sobiv õpetamiseks“, sai täidetud. Haskelli baasil funktsionaalprogrammeerimist õpetavaid raamatuid ja kursuseid leidub mitmeid, headeks näideteks S. Thompsoni „Haskell The Craft of Functional Programming“ [2], G. Hutton „Programming in Haskell“ ja Tartu Ülikooli kursus „Programmeerimiskeeled“ [1]. Ka eesti keeles leiduv H. Nestra raamat pealkirjaga „Sissejuhatus funktsionaalsesse programmeerimisse“ [4] baseerub just Haskellil.

Haskelli baasil saab väga hästi käsitleda eelnevalt defineeritud klassikalisi funktsionaalprogrammeerimise teemasid. Küll aga ei saa Haskellis otseselt käsitleda tänapäevaseid funktsionaalprogrammeerimise teemasid. Kuigi sõltuvalt tüübitud Haskelli arendatakse [8], on selle võimalused Haskelli pika ajaloo tõttu piiratud. Arenduse teeb keeruliseks nõue arvestada keele teatud omadustega, mis peavad erinevatel põhjustel säiluma, ning mida arenduse käigus lõhkuda ei saa. Seetõttu on mõistlik kaaluda alternatiivset keelt, mille olemus ja eesmärk on olnud algusest peale olla sõltuvate tüüpidega, kuid mis oleks samas ka sarnane Haskellile. Idris on Haskellist inspireeritud, sealhulgas selle süntaks, keele omadused ja ka suur hulk standardteekidest [9]. Seetõttu on Idrisel potentsiaal, et lisaks sõltuvate tüüpide otsesele käsitlusele, saab selles ilma suuremate kadudeta edasi anda ka senini enamasti Haskellil käsitletud teemasid.

2.2. Idris ja tüübipõhine arendus

Idris on sõltuvate tüüpidega puhas funktsionaalne programmeerimiskeel. Selle disainis, teostas ja arendab St. Andrews ülikooli lektor Edwin Brady ning see avaldati 2013. aastal [5]. Idrise hetkel uusim versioon on Idris 2, mis avaldati 2020. aastal ning baseerub kvantitatiivsel tüübiteoorial [6]. Idrise projekti eesmärk on ehitada sõltuvalt tüübitud keel, mis sobib tõestatava üldotstarbelise programmeerimise jaoks [5].

Edwin Brady argumenteerib, et tihti nähakse tüüpe pelgalt kui programmist vigade avastamise tööriista. Sellest mõjutatult kasutatakse tüübikontrolli abi sageli vaid alles valmis kirjutatud programmis vigade avastamiseks. Seda levinud arvamust saab muuta, tutvustades tüüpide ja tüübikontrolli kasulikkuse potentsiaali tüübipõhise arenduse abil. Tüübipõhisel arendusel Idrisega on tüübid tööriistaks hoopis programmi konstrueerimisel ning tüübikontroll assistent, mis programmeerijaid täieliku ja töötava programmi loomisel toetab [9]. Tüübipõhine arendus on üks Idrise keele alustalasid.

Erinevalt Haskellist, kus ei ole funktsiooni tüübi deklaratsiooni alati vaja välja kirjutada, on Idrises funktsiooni tüübi defineerimine möödapääsmatu [9]. Mida täpsemalt on defineeritud funktsiooni tüüp, seda lihtsam on vastava funktsiooni sisu (õigesti) kirjutada [12]. Seda, kuidas Idris funktsiooni tüübi definitsiooni abil selle sisu kirjutamisel abistada saab, vaadatakse peatükis 4.3.

Funktsiooni tüübi võimalikult täpset kirjapanekut võimaldavad Idrises sõltuvad tüübid, mis Haskellis sellisel kujul puuduvad. Sõltuv andmetüüp on tüüp, mis võib parameetriks võtta väärtuse. Klassikaline näide sõltuvast tüübist on etteantud pikkusega listid, kus tüüp sõltub listi pikkusest [5]. Sõltuvate tüüpide eeliseid kirjeldatakse peatükkides 4.1 kuni 4.3.

Võrreldes traditsiooniliste programmeerimiskeelte tüübisüsteemidega suurendavad sõltuvad tüübid tüübikontrolli ajal kontrollitavate omaduste hulka märkimisväärselt [10]. Sõltuvad tüübid annavad võimaluse väljendada valemeid tavaliste tüüpidena, kus tüüpidele vastavateks väärtusteks on valemite tõestused [10]. Tänu sõltuvatele tüüpidele saab Idrises funktsioone tõestada, mida Haskellis sellisel kujul teha ei saa. Valemite tüüpidena väljendamist tutvustatakse peatükis 4.4.

3. Klassikaliste teemade ülekandmine

Selles peatükis tehakse keeltevaheline analüüs kursusel „Programmeerimiskeeled" kaetavate teemade ja nendele vastavate ülesannete kontekstis. Töö käigus kohandati „Programmeerimiskeelte" kursuse [1] Haskelli osa praktikumiülesannete lahendused Idrisesse[2]. Kohandamise protsessi eesmärgiks oli kindlaks teha, kas praeguse kursuse praktikumide materjalid saab suuremate kadudeta uuele keelele üle kanda.

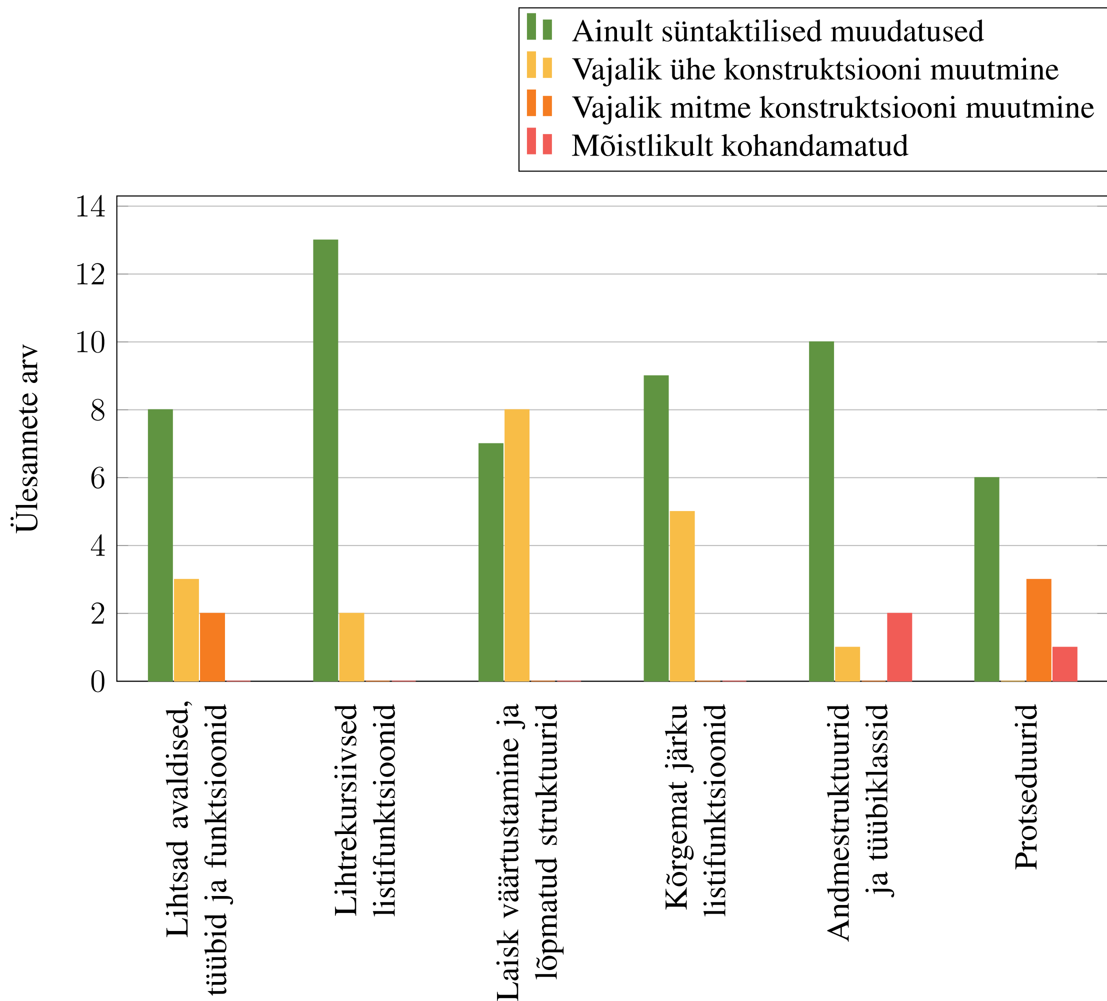
„Programmeerimiskeelte" aines on kuus Haskelli ülesannetel baseeruvat praktikumi. Igas praktikumis on omakorda näidis-, harjutus- ja tärnülesanded. Praktikumis käsitletavad teemad ja neile vastavate ülesannete arvud on toodud tabelis 1.

Tabel 1. Aine „Programmeerimiskeeled" Haskelli keelel baseeruvate ülesannete arvud praktikumide teemade ja ülesannete tüüpide kaupa.

#	Praktikumi teemad	Näidisülesandeid	Harjutusülesandeid	Tärnülesandeid
1	Lihtsad avaldised, tüübid ja funktsioonid	3	5	5
2	Lihtrekursiivsed listifunktsioonid	7	7	1
3	Laisk väärtustamine ja lõpmatud struktuurid	3	9	3
4	Kõrgemat järku listifunktsioonid	3	6	5
5	Andmestruktuurid ja tüübiklassid	3	8	2
6	Protseduurid	4	4	2

Kohandatud ülesanded on jaotatud tõlkimise keerukuse järgi nelja erinevasse kategooriasse, mis on nähtavad joonise 1 legendist. Järgnevates alampeatükkides kirjeldatakse protsessi käigus välja tulnud süntaktilisi, keelekonstruktsioonilisi kui ka otseselt keelte omadustest tingitud sarnasusi ja erinevusi koos neid illustreerivate ülesannete lahendustega.

[2] Kohandatud ülesannete kogu asub repositooriumis `https://bitbucket.org/karoliineh/idris-ulesannete-kogu`.



Joonis 1. Ülesannete kohandamisel ülesannetes tehtud muudatuste maht praktikumide kaupa.

3.1. Süntaksi erinevused

Jooniselt 1 on näha, et valdav osa kõikide teemade ülesannetest vajas vaid süntaktilist kohandamist. Lihtsamad ülesanded sai tõlkida vaid minimaalsete süntaktiliste erinevustega. Näiteks funktsiooni, mis leiab n-da Fibonacci arvu, saab defineerida Haskellis ja Idrises vastavalt:

Haskell

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib x = fib (x - 1) + fib (x - 2)
```

Idris

```
fib : Int -> Int
fib 0 = 0
fib 1 = 1
fib x = fib (x - 1) + fib (x - 2)
```

Funktsiooni `fib` keeltevaheline erinevus seisneb vaid ühes väikeses süntaksi erinevuses. Haskellis kasutatakse funktsiooni tüübi defineerimiseks funktsiooni nime taga topelt koolonit, Idrises kasutatakse samal eesmärgil samas kohas aga ühekordset koolonit. Kõik muu nende funktsioonide juures on identne: funktsiooni tüüpe tähistatakse noolega `->`, mõlemas keeles saab teha mustrisobitust ning ka funktsiooni rakendamise süntaks on sama.

Põhilised kaks listidega seotud süntaksierinevust tulid välja ülesandes, kus tuleb luua funktsioon, mis arvutab täisarvude listi summa. See funktsioon on defineeritud Haskellis ja Idrises vastavalt:

Haskell

```haskell
sum :: [Integer] -> Integer
sum []        = 0
sum (x : xs) = x + sum xs
```

Idris

```idris
sum : List Integer -> Integer
sum []          = 0
sum (x :: xs) = x + sum xs
```

Funktsiooni `sum` definitsioonidest on näha, et ühe elemendi listi algusesse lisamise infix funktsiooni tähis on Haskellis ühekordne koolon, kuid Idrises on see vastupidiselt kahekordne koolon. Lisaks tähistatakse erinevalt listi tüüpi: Haskellis tähistatakse listi kahe nurksuluga, mille vahele on kirjutatud listis olevate elementide tüüp, Idrises aga tähistatakse listi võtmesõnaga `List`, millele järgneb listi elementide tüüp.

Üsna suures osas erines ühe konstruktori ja mitme väljaga tüüpide defineerimise süntaks. Vastav erinevus kajastus ülesandes, kus tuli defineerida tüüp `Kiri` konstruktoriga `K`, mis täidab väljad: `saatja, saaja, pealkiri, sisu`.

Haskell

```haskell
data Kiri = K {
  saatja   :: Email,
  saajad   :: [Email],
  pealkiri :: String,
  sisu     :: String
}  deriving (Show)
```

Idris

```idris
record Kiri where
  constructor K
  saatja   : Email
  saajad   : List Email
  pealkiri : String
  sisu     : String
```

Idrises puudub Haskelli võtmesõnale `deriving` vastav direktiiv. Haskellis annab süntaks `deriving (Show)` kompilaatorile märku, et see genereeriks vastavale andmetüübile automaatselt tüübiklassi `Show` teostuse. `Show` on prelüüdis defineeritud tüübiklass, mis pakub võimalust defineerida, kuidas peaks erinevat tüüpi väärtuseid sõne kujul kuvama.

Idrises on lihtsaim alternatiiv uuele andmetüübile `Show` tüübiklass manuaalselt teostada. Teine võimalus on tüübiklasside tuletamine realiseerida Idrise metaprogrammeerimise teel. Idrise kogukonna poolt on loodud repositooriume [11], mis seda teevad, kuid Idrise põhiprojektis pole tüübiklasside tuletamine veel teostatud.

Idrises, nagu ka Haskellis, on olemas monaadiline `do`-notatsioon. Ka nende süntaksid on sarnased, välja arvatud üks võtmesõna erinevus: Idrises kasutatakse Haskelli võtmesõna `return` asemel võtmesõna `pure`. Näitena on toodud lahendus ülesandele luua funktsioon nimega `prindiArvud1`, mis prindib argumendina antud arvude listis olevad arvud järjestikku eraldi ridadele.

Haskell

```
prindiArvud1 :: [Int] -> IO ()
prindiArvud1 [] = return ()
prindiArvud1 (x:xs) = do
  print x
  prindiArvud1 xs
```

Idris

```
prindiArvud1 : List Int -> IO ()
prindiArvud1 [] = pure ()
prindiArvud1 (x::xs) = do
  printLn x
  prindiArvud1 xs
```

3.2. Keelekonstruktsioonide erinevused

Osade ülesannete lahendustes pidi lisaks süntaktilisele kohandamisele muutma või lisama keelekonstruktsioone. Joonisel 1 on ülesanded, mis vajasid kas ühe või mitme konstruktsiooni muutmist, kujutatud vastavalt kollaselt ja oranžilt. Selles alampeatükis on kirjeldatud põhilised Haskelli ja Idrise keelekonstruktsioonide erinevused koos näidetega, kuidas need mõjutasid ülesannete ehitust.

3.2.1. Sõned

Erinevalt Haskellist, kus on sõne kujutatud märkide listina, on Idrises sõne primitiivne andmetüüp [9]. Kui Haskellis kehtib avaldis `['a','b','c'] == "abc"` ehk märkide list on sõne sünonüüm, siis Idrises see võrdus ei kehti ehk märkide list ei ole sõne sünonüüm. Idrises tuleb sõnede ja märgilistide vahel opereerimiseks, kas sõne märkide listiks lahti pakkida või vastupidi märkide list sõneks pakkida. Sellist pakkimist pidi kasutama ülesandes, kus tuli kirjutada funktsioon `lines`, mis võtab sisse sõne ja tagastab sõnede listi, mille elemendid on sisendi tekstiread. Sisendsõnes on reavahetus tähistatud reavahetusmärgiga, mistõttu on vaja sõne töödelda kui märkide listi.

Haskell

```haskell
lines :: String -> [String]
lines ""          = []
lines ('\n':xs) = "" : lines xs
lines (x:xs)     =
  case lines xs of
    []      -> [[x]]
    (l:ls) -> (x : l) : ls
```

Idris

```idris
lines : String -> List String
lines s = map pack (lines' (unpack s))
  where
    lines' : List Char -> List (List Char)
    lines' []           = []
    lines' ('\n'::xs) = [] :: lines' xs
    lines' (x::xs)     =
      case lines' xs of
        []       => [[x]]
        (l::ls) => (x :: l) :: ls
```

Samal põhjusel kaotas aga üks ülesanne täielikult oma mõtte. Ülesandeks oli kirjutada funktsioon `upper`, mis võtab argumendiks sõne ning muudab selle kõik tähed suurtähtedeks, kasutades selleks standardpreluüdi funktsiooni `toUpper`. Haskellis on funktsioon `toUpper` tüübiga `Char -> Char`. Idrise standarpreluüdis on aga olemas samanimeline funktsioon tüübiga `String -> String`. Seega funktsioon `toUpper` Idrises täpselt seda ülesannet lahendabki ning ülesande sisuks jääks funktsioon lihtsalt ümber nimetada.

Haskell

```haskell
upper :: String -> String
upper = map toUpper
```

Idris

```idris
upper : String -> String
upper = toUpper
```

3.2.2. Valvurid

Idrises puudub analoog Haskellis kasutatavale valvurite süntaksile. Selle asemel saab kasutada näiteks `where` ja `case of` või `with` konstruktsioone. Küll aga ei ole nende kummagi kirjapilt kindlasti nii elegantne, kui valvuritega koodi kirjapilt, vaid meenutavad pigem pesastatud `if`-lausete jada. Näiteks aine „Programmeerimiskeeled" hinde arvutamise funktsioon on Haskellis:

```haskell
hinne :: Bool -> Bool -> Int -> Int -> Int -> Char
hinne True True kodu loeng eks
  | s >= 90   = 'A'
  | s >= 80   = 'B'
  | s >= 70   = 'C'
  | s >= 60   = 'D'
  | s >= 50   = 'E'
  | otherwise = 'F'
  where s = kodu + loeng + eks
hinne hb sb kodu loeng eks = 'F'
```

Sama funktsioon Idrises alternatiivsete `where` ja `case of` konstruktsioonidega on aga:

```
hinne : Bool -> Bool -> Int -> Int -> Int -> Char
hinne True True kodu loeng eks = arvuta (kodu + loeng + eks)
  where
    arvuta : Int -> Char
    arvuta s =
      case (s >= 90) of
        True  => 'A'
        False => case (s >= 80) of
          True  => 'B'
          False => case (s >= 70) of
            True  => 'C'
            False => case (s >= 60) of
              True  => 'D'
              False => case (s >= 50) of
                True  => 'E'
                False => 'F'
hinne hb sb kodu loeng eks = 'F'
```

Funktsioonides, kus on vaja kasutada vaid kahte teineteist välistavat valvurit, saab need asendada `with` konstruktsiooniga ning seejärel otsustada, mida teha `True` ja `False` korral.

Haskell

```
removeAll1 :: Int -> [Int] -> [Int]
removeAll1 n = foldr (#) []
  where
    x # acc
      | x == n     = acc
      | otherwise = x : acc
```

Idris

```
removeAll1 : Int -> List Int -> List Int
removeAll1 n = foldr (#) []
  where
    x # acc with (x == n)
      | True  = acc
      | False = x :: acc
```

Funktsioon `removeAll1` võtab argumentideks arvu ja arvude listi ning tagastab listi, mis on saadud teisest argumendist kõigi esimese argumendi esinemiste ära jätmisel. Küll aga tuleb `with` konstruktsiooni kasutamisega olla ettevaatlik, kuna 2021. aasta kevade seisuga on see Idris 2s veel puudu [6].

3.2.3. Laiskusega seotud probleemid

„Programmeerimiskeelte“ kursuses oli laiskusele pühendatud terve kolmas Haskelli praktikum, mille teemadeks olid laisk väärtustamine ja lõpmatud struktuurid. Haskelli laiskusest ja Idrise agarusest tingituna nõuavad need teemad aga mõningaid muudatusi.

S. Thompson kirjeldab, et laisk väärtustaja väärtustab argumente vaid siis, kui funktsioon neid enda väärtuse arvutamise jaoks kasutab. Kui funktsiooni argumendiks on andmestruktuur, näiteks list või ennik, väärtustatakse ka sellest ainult osa, mida funktsioon arvutamiseks vajab [2]. Seepärast on Haskellis olemas lõputud andmestruktuurid, millele on laiskuse tõttu võimalik funktsioone rakendada ilma, et need lõpmatult jooksma jääksid.

Erinevalt Haskellist, mis on laisa väärtustamisega, on Idris agara (*strict*) väärtustamisega. Agar väärtustamine on laisa väärtustamise vastand ja tähendab, et enne funktsiooni väärtuse arvutamist tema argumendid igal juhul väärtustatakse [4]. Idrise dokumentatsioonis on kirjas, et Idris kasutab agarat väärtustamist jõudluse paremaks prognoosimiseks. Täpsemalt seetõttu, et pikema perspektiivi eesmärk on võimaldada efektiivse verifitseeritud madala taseme koodi kirjutamist näiteks seadmete draiverite ja võrgu infrastruktuuride jaoks [5].

Kuigi Idrises on programmi töö ajal väärtustamine agar, on selles võimalik laiskus saavutada laisa tüübi (*lazy type*) abil. Näiteks saab Idrises kirjutada laisa `foldr` funktsiooni, milleks on vähemalt kaks erinevat võimalust:

```
1  foldr : (a -> Lazy b -> b) -> b -> List a -> b
2  foldr _ a []      = a
3  foldr f a (x::xs) = f x (foldr f a xs)
4
5  data LazyList : Type -> Type where
6    Nil  : LazyList a
7    (::) : (x : a) -> (xs : Lazy (LazyList a)) -> LazyList a
8
9  foldrLazy : (a -> Lazy b -> b) -> Lazy b -> LazyList a -> b
10 foldrLazy _ acc []      = acc
11 foldrLazy f acc (x::xs) = f x (foldrLazy f acc xs)
```

Lõpmatute listidega ülesanded on Idrises lahendatavad tüübiga `Stream`, mis on lõpmatu laisk list. Näiteks on toodud funktsioon `fibs`, mis leiab lõpmatu Fibonacci arvude listi.

Haskell
```
fibs :: [Integer]
fibs = fibs' 0 1
  where
    fibs' a b = a : fibs' b (a + b)
```

Idris
```
fibs : Stream Int
fibs = fibs' 0 1
  where
    fibs' : Int -> Int -> Stream Int
    fibs' a b = a :: fibs' b (a + b)
```

3.2.4. Erindid

Idrises ei saa visata erindeid nii, nagu seda saab Haskellis. Seega näiteks ülesandes, kus peab leidma listi viimase elemendi, tuleb tühja listi korral erindi viskamise asemel Idrises kasutada `Maybe` keelekonstruktsiooni. Selleks peab tagastuastüübi `a` asendama tüübiga `Maybe a`, mille konstruktorid on `Just` ja `Nothing`, millest viimane asendab erindi viskamist.

Haskell
```
last :: [a] -> a
last []      = error "last: tühi list"
last [x]     = x
last (_:xs)  = last xs
```

Idris
```
last : List a -> Maybe a
last []       = Nothing
last [x]      = Just x
last (_::xs)  = last xs
```

3.2.5. Tüübiklassid

Haskelli tüübiklasside (*type classes*) analoogiks Idrises on liidesed (*interfaces*), kuid neil on mõningad erinevused. Tähtsaim erinevus on, et Idrise liideseid saab parametriseerida *ükskõik* mis tüüpi väärtustega, näiteks täisarvu tüüpi parameetriga, mis Haskellis lubatud ei ole [9]. Teiseks erinevuseks on, et Idrise liidestel on lubatud mitu teostust [9]. Idrise dokumentatsioonis [5] on kirjas, et ühele liidesele mitme teostuse loomist võib kasutada näiteks juhul, kui tahetakse kasutada alternatiivseid sortimis- või printimismeetodeid. Selleks saab anda liideste teostustele nimed. Näiteks on toodud naturaalarvude võrdlemise liidese ümberpööratud teostus, millele on antud nimi `myRevOrd`:

```
[myRevOrd] Ord Nat where
   compare Z (S n)      = GT
   compare (S n) Z      = LT
   compare Z Z          = EQ
   compare (S x) (S y) = compare @{myRevOrd} x y
```

Eelnev defineerib liidese nagu tavaliselt, kuid annab sellele lisaks kindla nime `myRevOrd`. Süntaks `@{myRevOrd}` näitab, et tuleb kasutada just sellenimelist liidese teostust, vastasel juhul kasutatakse standardteegis defineeritud ilma nimeta `Ord` liidese teostust. Kui rakendada mõnele naturaalarvude listile `sort` meetodit, kasutatakse vaikimisi `Ord` liidese teostust ning list sorteeritakse kasvavalt. Kui aga rakendada samale naturaalarvude listile sama meetodit, kasutades eelnevalt defineeritud nimelise liidese teostust `sort @{myRevOrd}`, sorteeritakse list kahanevalt [5].

Nimega liideseid saab kasutada ka nende laiendustes. Näiteks defineeritakse prelüüdis liides `Semigroup`, mida realiseerivatel tüüpidel tuleb defineerida üks binaarne assotsiatiivne operaator <+>:

```
interface Semigroup ty where
  (<+>) : ty -> ty -> ty
```

Seejärel on defineeritud `Monoid`, mis laiendab `Semigroup` liidest ühikelemendiga:

```
interface Semigroup ty => Monoid ty where
  neutral : ty
```

Dokumentatsioonis on näidatud, kuidas saab defineerida tüübile `Nat` kaks erinevat `Semigroup` ja `Monoid` teostust, kus üks baseerub naturaalarvude liitmisel ja teine korrutamisel:

```
[PlusNatSemi] Semigroup Nat where
  (<+>) x y = x + y

[MultNatSemi] Semigroup Nat where
  (<+>) x y = x * y

[PlusNatMonoid] Monoid Nat using PlusNatSemi where
  neutral = 0

[MultNatMonoid] Monoid Nat using MultNatSemi where
  neutral = 1
```

Liitmise ühikelement on 0, aga korrutamise ühikelement on 1. Seega on oluline, et vastavad `Monoid` liidesed laiendavad neile vastavaid `Semigroup` teostuseid. Selleks on kasutatud `using` võtmesõna, millele järgneb vastava `Semigroup` liidese nimi [5].

3.3. Mõistlikult kohandamatud ülesanded

Joonisel 1 on punasega kujutatud ülesanded, mida polnud võimalik mõistlikult kohandada ning mille kohandamine ei olnud töö eesmärgi täitmiseks vajalik. Nende hulka kuuluvad viienda praktikumi tärnülesanded ning üks kuuenda praktikumi tärnülesanne:

1. Kirjuta `Semigroup` ja `Monoid` tüübiklasside instantsid `Mat22` tüübile nii, et `(<>)` operaator korrutaks maatriksid ja `mempty` oleks ühikmaatriks. Korrektse lahenduse korral saab n-inda Fibonacci arvu arvutada efektiivselt keerukusega $O(\log(n))$, kasutades ülesandes defineeritud maatriksite korrutamise monoidi.

2. Kirjuta `Num` tüübiklassi instants `Mat22` tüübile. Korrektse lahenduse korral saab n-inda Fibonacci arvu arvutada efektiivselt keerukusega $O(\log(n))$, kasutades ülesandes defineeritud maatriksite aritmeetikat.

3. Rekursiivne kataloogide läbimine. Kasutades funktsioone moodulist `System.Directory`, implementeerige rekursiivne kataloogi suuruse arvutamise protseduur. Faili suuruse arvutamine teha ette antud funktsiooniga failiSuurus. S.t kataloogi suurusena loeme selles olevate failide suuruste summa pluss alamkataloogide suurus.

„Programmeerimiskeelte“ viienda praktikumi tärnülesannete sisuks on tüübiklasside sügavamalt tundma õppimine. Konkreetsete ülesannete Idrises lahendamise teeb keeruliseks Haskelli standardprelüüdis oleva meetodi `stimesMonoid` ja `Num` liidesel baseeruva astendamise puudumine Idrise standardprelüüdis. Seega pidi Idrises lisaks ülesannete lahendustele teostama ka puuduvad funktsioonid. Eelnevast tulenevalt on lahendused kordades pikemad kui Haskellis. Sellisest pikast lahenduskäigust aru saamine ei täida ülesande eesmärki ning siinkohal on mõistlik kaaluda ülesannetele alternatiive.

Kataloogide rekursiivse läbimise ülesanne eeldab spetsiifilise Haskelli mooduli `System.Directory` kasutamist, milles olevaid kataloogi läbimise funktsioone Idrise standardteegis ei ole. Leidub Idrise teeke, mille põhjal sellist ülesannet lahendada saaks, kuid praktikumi eesmärk on õppida `IO` monaade, mitte lahendada konkreetset ülesannet. Seega on mõistlik ka siin kaaluda ülesande asendamist.

3.4. Klassikaliste teemade käsitlemine Idrises

Analüüsides kolmanda peatüki sisu, saab sõnastada vastuse püstitatud uurimisküsimuse esimesele poolele: Kuidas mõjutab Haskelli asendamine Idrisega klassikaliste funktsionaalprogrammeerimise teemade käsitlemist?

Praktikumiülesannete kohandamise käigus selgus, et keele vahetamise mõju klassikaliste funktsionaalprogrammeerimise teemade õpetamisele on väike, sest enamikke ülesandeid sai Idrises lahendada. Sealhulgas tuli välja, et suurema osa ülesannetest sai tõlkida vaid süntaktiliste erinevustega, kuid leidus ka ülesandeid, mida mõistlikult kohandada ei olnud võimalik. Seega tuleb mõndade ülesannete osas kaaluda nende asendamist ülesannetega, mille lahendused on Idrises elgentsemalt väljendatavad.

4. Sõltuvate tüüpidega programmeerimine ja verifitseerimine

Selle peatüki eesmärk on anda ülevaade teemadest, mille saab Idrisele üle minnes uute teemadena kursuse kavva võtta. Uute teemade valikus on lähtutud eesmärgist näidata, kuidas tugevam tüübisüsteem võib olla abiks programmide arendamisel ja verifitseerimisel.

4.1. Tüüpide arvutamine ja sõltuvate tüüpidega funktsioonid

Idrises on tüübid esimest klassi, mis tähendab, et tüübid ja väärtused on fundamentaalselt samad. Tüüpe saab arvutada, manipuleerida ja funktsioonidele argumentideks anda täpselt nagu ükskõik milliseid teisi keelekonstruktsioone [5]. Seega on tüübitaseme funktsioonide kirjutamine Idrises sama nagu mistahes teise andmetaseme funktsioonide kirjutamine. Idrise dokumentatsiooni [5] näidete põhjal antakse intuitsioon, kuidas tüüpe arvutatakse ning kus ja kuidas neid funktsioone kasutada saab. Dokumentatsiooni näidetes on antud funktsioon `isSingleton`, mis arvutab ja tagastab tüübi:

```
isSingleton : Bool -> Type
isSingleton True  = Nat
isSingleton False = List Nat
```

See funktsioon võtab argumendina tõeväärtuse, mis tähistab, kas tüübiks on üheelemendiline hulk. Kui funktsiooni argumendi väärtus on `True`, on tegu ühe elemendiga ning tagastatakse tüüp `Nat` ehk naturaalarv. Kui funktsiooni argumendi väärtus on aga `False`, tagastatakse tüüp `List Nat` ehk list naturaalarvudest.

Tüüpe tagastavaid funktsioone saab kasutada ükskõik, kus tüüpe kasutada saab. Seda illustreerib näide, kus funktsiooni `isSingleton` kasutatakse funktsiooni `initial` tagastustüübi arvutamiseks:

```
initial : (x : Bool) -> isSingleton x
initial True  = 0
initial False = []
```

Selles funktsioonis tagastatakse vastavalt argumendi `x` väärtusele, kas `Nat` tüüpi väärtus `0` või `List Nat` tüüpi väärtus `[]`. Samuti saab funktsiooni `isSingleton` kasutada ka

funktsiooni sisendtüübi arvutamiseks, mille illustreerivaks näiteks on funktsioon `sum`:

```
sum : (single : Bool) -> isSingleton single -> Nat
sum True  x  = x
sum False [] = 0
sum False (x :: xs) = x + sum False xs
```

Funktsioon `sum` arvutab, kas naturaalarvudest listi elementide summa või tagastab lihtsalt sisendiks olnud naturaalarvu vastavalt sellele, kas argumendiks on list või naturaalarv [5]. Olles nüüd tutvunud tüüpide arvutamise kontseptsiooniga, saab defineerida näiteks kahedimensioonilise punkti tüübi:

```
punkt2d : Type
punkt2d = (Double, Double)
```

ning seda üldistades defineerida funktsiooni, mis arvutab sõltuvalt naturaalarvulisest sisendist sellemõõtmelisse dimensiooni kuuluva punkti tüübi:

```
punkt : Nat -> Type
punkt Z     = Unit
punkt (S Z) = Double
punkt (S n) = (Double, punkt n)
```

Sisendi `0` ehk `Z` korral on funktsiooni tagastustüüp `Unit` ehk tühi tüüp. Sisendiga `1` ehk `(S Z)` tagastab funktsioon tüübi `Double`, sest 1-dimensioonilises ruumis saab punktil olla vaid üks koordinaat. Ühest suurema naturaalarvu `(S n)` korral tehakse rekursiivne funktsiooni kutse ühe võrra väiksema sisendväärtusega, lisades igal kutsel ühe koordinaadi koha juurde seni, kuni rekursioon ühe juures termineerub. Seega tagastab funktsioon sisendi `2` korral tüübi `(Double, Double)`, `3` korral tüübi `(Double, (Double, Double))` jne. Siinkohal on oluline teada, et sisemiselt hoiab Idris kõiki ennikuid peale tühja enniku pesastatud paaridena ehk `(Double, (Double, Double)) == (Double, Double, Double)`.

Nii funktsiooni `isSingleton` kui ka `punkt` korral on tegu sõltuvate tüüpidega funktsioonidega, sest tagastatav tüüp sõltub sisendi väärtusest. Funktsiooniga `punkt` saab näiteks defineerida kahe erineva punkti tüübid: `xy`, mis on 2-dimensioonilises ruumis, ja `xyz`, mis on 3-dimensioonilises ruumis:

```
xy : punkt 2
xy = (2.0,4.0)

xyz : punkt 3
xyz = (3.0,6.0,3.0)
```

Tüüpi arvutava funktsiooni `punkt` põhjal saab näiteks koostada kolm harjutusülesannet, milles tüüpi arvutava funktsiooni kasutamist harjutada.

Ülesanne 1 - `nullpunkt` - **erinevate dimensioonide nullpunktide arvutamine**

Kirjuta funktsioon, mis arvutab vastavalt naturaalarvulisele sisendile sellelemõõtmelise dimensiooni nullpunkti:

```
nullpunkt : (d : Nat) -> punkt d
nullpunkt d = ?undefined
```

Näiteks:

```
nullpunkt 0 ==> ()
nullpunkt 1 ==> 0.0
nullpunkt 2 ==> (0.0, 0.0)
```

Lahendus - `nullpunkt`

```
nullpunkt : (d : Nat) -> punkt d
nullpunkt Z         = ()
nullpunkt (S Z)     = 0.0
nullpunkt (S (S n)) = (0.0, nullpunkt (S n))
```

Ülesanne 2 - `add` - **liida kahe punkti vastavad koordinaadid**

```
add : (d : Nat) -> punkt d -> punkt d  -> punkt d
add d x y = ?add_rhs
```

Näiteks:

```
add 0 () () ==> ()
add 1 1.0 2.0 ==> 3.0
add 3 (0.2, 0.2, 4.0) (1.0, 2.0, 0.2) ==> (1.2, 2.2, 4.2)
```

Lahendus - `add`

```
add : (d : Nat) -> punkt d -> punkt d  -> punkt d
add Z _ _     = ()
add (S Z) x y = x + y
add (S (S n)) (x,xs) (y,ys) = (x + y, add (S n) xs ys)
```

Ülesanne 3 - `sum` - **liida kõigi listis olevate punktide koordinaadid**

Kirjuta funktsioon, mis võtab sisse naturaalarvu ja sellele vastavasse dimensiooni kuuluvate punktide listi ning tagastab punkti, mille koordinaatideks on teise argumendina saadud listis olevate punktide vastavate koordinaatide summa, kasutades selleks eeldefineeritud funktsioone `nullpunkt` ja `add`:

```
sum : (d : Nat) -> List (punkt d) -> punkt d
sum d xs = ?sum_rhs
```

Näiteks:

```
sum 0 [] ==> ()
sum 2 [(1.0,1.0), (2.0,2.0)] ==> (3.0, 3.0)
```

Lahendus - `sum`

```
sum : (d : Nat) -> List (punkt d) -> punkt d
sum d []      = nullpunkt d
sum d (x::xs) = add d x (sum d xs)
```

Kõikides funktsioonides, mis kasutavad enda definitsioonis funktsiooni `punkt`, pidi ühe lisaparameetrina kaasas kandma naturaalarvu, mis tähistas vastava punkti koordinaatide arvu ehk dimensiooni, milles need punktid asuvad. Sõltuvas tüübis saab selle arvu aga juba tüübi enda sisse panna.

4.2. Sõltuvad tüübid

Lisaks sellele, et esimest klassi tüüpidega saab kirjutada funktsioone, mis arvutavad tüüpe sõltuvalt nende sisendi väärtusest, võimaldavad need lisada väärtuseid ka tüübi enda kirjeldusse [5]. See muudab tüübid väljendusrikkamaks ja seega ka täpsemaks.

4.2.1. Vektorid

Kõige klassikalisem näide Idrise sõltuvatest tüüpidest on vektorid. Vektorid on listid, mis on parametriseeritud nende pikkuse suhtes. See tähendab, et lisaks listile endale sisaldub tüübis alati ka selle pikkus [5, 13]. Üldine tüüp `List` on Idrise standardteegis defineeritud kui:

```
1 data List : (a : Type) -> Type where
2   Nil  : List a
3   (::) : a -> List a -> List a
```

Vektori üldine tüüp on samuti osa Idrise standardteegist ja on defineeritud kui:

```
1 data Vect : (k : Nat) -> (a : Type) -> Type where
2   Nil  : Vect Z a
3   (::) : a -> Vect k a -> Vect (S k) a
```

Dokumentatsiooni põhjal deklareerib eelnev definitsioon tüüpide perekonna ning seetõttu on selle kuju erinev peatükis 4.1 nähtud tüüpi arvutavast funktsioonist. Tüübikonstruktor `Vect` võtab argumendina sisse naturaalarvu `k : Nat` ja tüübi `a : Type`, millest viimase tüüp on tüüpide tüübiks. Öeldakse, et `Vect` on üle naturaalarvude indekseeritud ja tüübi `a` poolt parametriseeritud [5, 9]. Viimane tähendab, et indeks võib üle struktuuri muutuda ehk vektori igal alamvektoril saab olla erinev pikkus, aga andmestruktuuri elementide tüübid terves selle ulatuses muutuda ei tohi.

Iga konstruktor loob erineva osa tüübi perekonnast. Konstruktor `Nil` saab luua vaid vektoreid pikkusega null ja konstruktor `::` vektoreid pikkusega rohkem kui null. Konstrukturi `Nil` poolt konstrueeritav tüüp näitabki, et seda tüüpi vektor peab olema pikkusega `Z` (*zero*), mis vastab naturaalarvu väärtusele null. Konstruktori `::` tüübis kirjeldatakse, et elemendid `a` ja `Vect k a`, teisisõnu vektor pikkusega `k`, annavad omavahel kombineeritult vektori pikkusega `S k` [5]. `S k` ehk `k` järeltulijat (*successor of*

k) saab lugeda kui $k + 1$.

Kui tüübid saavad sisaldada väärtuseid, mis kirjeldavad omadusi, saab funktsiooni tüübis kirjeldada funktsiooni enda omadusi. Näiteks kahe listi konkateneerimise funktsioonil on omadus, et tulemuseks oleva listi pikkus on kahe sisendiks oleva listi pikkuste summa. Kahe vektori konkateneerimise funktsioon on seega defineeritud kui [5, 9]:

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append []        ys = ys
append (x :: xs) ys = x :: append xs ys
```

Kuna tüübid on esimest klassi, on tüübid ja avaldised osa samast keelest [9]. Seega saab funktsiooni tüübi definitsioonis kasutada naturaalarvude `n` ja `m` peal ükskõik millist naturaalarvude avaldist, praegusel juhul liitmistehet [9].

Edwin Brady argumenteerib, et tihti kirjutab ta Haskellis funktsioone eeldades, et list ei ole tühi, ning seejärel mitu kuud hiljem rikub seda eeldust mõnes muus failis [13]. Vektorid pakuvad sellistele juhtumitele hea lahenduse. Lisaks annavad vektorid võimaluse välistada juhud, kus listide pikkused ei ühti. Seega saab vektoreid kasutada selliste funktsioonide defineerimisel, kus tahetakse eeldada, et funktsioon eripikkuste listide peal töötada ei tohiks ning seega välistada, et funktsiooni eripikkuste listide peal üldse rakendada saaks.

Listi pikkust saab käsitleda ka kui selle kuju. Andmestruktuuri kujuks on andmestruktuuri tekitatav vari [12]:

```
[]    [1]    [1, 2]     [1, 2, 3]
[]    [■]    [■, ■]     [■, ■, ■]
Z     S k    S (S k)    S (S (S k))
```

Listi kujuks on selle pikkus, milleks on naturaalarv. Sarnase mõttekäiguga saab luua ka sõltuva tüübiga puud, mille tüüp sõltub puu kujust.

4.2.2. Kujulised puud

Tallinna Tehnikaülikooli kursusel „Funktsionaalprogrammeerimine“ tõi teadur E. B. Morehouse sõltuvate tüüpide teemas sisse vektoritest keerulisema näite, mida nimetas terminiga *shapely trees*, otsetõlkes kujulised puud [12]. Kui listi kuju vektoris oli

defineeritud kui naturaalarv, siis puu kuju on defineeritud kui:

```
data TreeShape : Type where
   LeafShape : TreeShape
   NodeShape : (l : TreeShape) -> (r : TreeShape) -> TreeShape
```

ning kujulised ehk üle kuju indekseeritud puud on defineeritud kui:

```
data Tree : TreeShape -> Type -> Type where
   Leaf : Tree LeafShape a
   Node : (left : Tree l a) -> (this : a) -> (right : Tree r a) ->
          Tree (NodeShape l r) a
```

Morehouse argumenteerib, et kujuliste puudega erinevate operatsioonide, näiteks kahe puu üheks kokku pakkimise (*zip*), tipu asendamise, oksa pookimise ja puude konkateneerimise funktsioonide kood lihtsustub kordades, kui puu on üle oma kuju indekseeritud [12]. Näiteks tavalise puu, defineeritud kui:

```
data Tree : Type -> Type where
   Leaf : Tree a
   Node : Tree a -> a -> Tree a -> Tree a
```

korral on kahe puu kokkupakkimiseks defineeritud funktsioon:

```
zip_tree : Tree a -> Tree b -> Tree (Pair a b)
zip_tree Leaf Leaf  =  Leaf
zip_tree Leaf (Node left this right)  =  Leaf
zip_tree (Node left this right) Leaf  =  Leaf
zip_tree (Node left1 this1 right1) (Node left2 this2 right2)  =
   Node (zip_tree left1 left2) (this1 , this2) (zip_tree right1 right2)
```

Funktsiooni juhte ridadel 3 ja 4 ei tohiks tegelikult täide viia: ei tohiks juhtuda, et ühes puus jõutakse lehttippu, kuid teises puus on samas tipus veel läbimata alampuid. Kujulise puu korral on aga juba enne funktsiooni rakendamist teada, kas kokkupakitavad puud on sama kujuga või mitte. Seega ei saa lubamatud juhud ette tulla ning ka funktsiooni kirjapilt on lühem ja selgem:

```
zip_tree : Tree shape a -> Tree shape b -> Tree shape (Pair a b)
zip_tree Leaf Leaf =  Leaf
zip_tree (Node left1 this1 right1) (Node left2 this2 right2)  =
   Node (zip_tree left1 left2) (this1 , this2) (zip_tree right1 right2)
```

Kuigi koodi kirjapilti lihtsustava efekti saab kujuta puudel luua vaikimisi juhuga:

```
zip_tree : Tree a -> Tree b -> Tree (Pair a b)
zip_tree (Node left1 this1 right1) (Node left2 this2 right2)  =
   Node (zip_tree left1 left2) (this1 , this2) (zip_tree right1 right2)
zip_tree _ _  =  Leaf
```

siis ei välista see juhte, kus tõesti ei taheta, et kaks erikujulist puud kokku pakitaks. Nimetatud funktsiooniga pakitakse puud kokku viisil, kus tulemuseks on puu, mille kuju on argumentideks antud puude kujude suurim ühisosa. Alternatiiv on kujuta puude soovitamatute juhtude korral visata erind ning sellega vajadusel edasi tegeleda. Idrise tüübisüsteem annab aga võimaluse funktsiooni mitte üldse rakendada lasta ehk viga ilmneb enne funktsiooni rakendamist, mitte alles siis, kui funktsioon on puus jõudnud kohta, kus kujud ei klapi.

Tüübisüsteemi võimekust vigadest teada anda juba ainuüksi funktsiooni rakendada üritades, aitab illustreerida kujuliste puude peal defineeritud funktsioon, mis peegeldab puu, vahetades selles omavahel kõikide tippude vasakud ja paremad alampuud.

```
flip_shape : TreeShape -> TreeShape
flip_shape LeafShape = LeafShape
flip_shape (NodeShape l r) = (NodeShape (flip_shape r) (flip_shape l))
flip_tree : Tree shape a -> Tree (flip_shape shape) a
flip_tree Leaf = Leaf
flip_tree (Node left this right) = Node (flip_tree right) this (flip_tree
    left)
```

Proovides nüüd rakendada mittetasakaalustatud puul `t` funktsiooni, kus üritatakse selle peegeldused omavahel kokku pakkida: `zip_tree (flip_tree t) (flip_tree t)`, on tüübikontroll rakendusega rahul ning kuvatakse tulemus. Küll aga kui proovida kokku pakkida esialgne puu `t` selle peegeldusega: `zip_tree t (flip_tree t)`, annab REPL teada, et tüübid ei kattu ning funktsiooni pole seetõttu võimalik rakendada.

4.3. Interaktiivne programmeerimine

Mõnikord ei ole funktsiooni kirjutamisel kohe selge, kuidas seda täpselt kirjutada nii, et see teeks just seda, mida tahetakse. Idris kergendab seda vaeva mitmete abistavate tegevustega. Nende hulka kuuluvad näiteks funktsiooni definitsiooni skeleti andmine, juhtudeks jagamine (*case split*), muutujatele vastavate tüüpide küsimine ja aukude kasutamine.

Eelnimetatud tegevustele vastab hulk Idrise REPLis sisalduvaid käske. Käsud töötavad kindlal koodi real oleval kindlal nimel, andes vastuseks sellekohast infot või uue programmifragmendi [5]. Tekstiredaktorid saavad vastavaid redigeerimiskäske pakkudes võimaldada interaktiivse programmeerimise tuge. Idrisele interaktiivse programmeerimise tuge pakkuvad tekstiredaktorid on näiteks Atom, Vim ja Emacs. Järgnevalt vaadatakse, mida kirjeldatud tegevused ja neile vastavad käsud täpselt teevad.

4.3.1. Funktsiooni definitsiooni skeleti andmine

Käsuga `:addclause` saab lasta Idrisel lähtuvalt funktsiooni tüübist pakkuda funktsiooni definitsiooni skeleti. Olgu defineeritud kahe vektori kokku pakkimise funktsiooni tüüp:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
```

Sellel real käsu `:addclause` kasutamine loob funktsiooni definitsiooni skeleti:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
zip_vect x y = ?zip_vect_rhs
```

Definitsiooni skeleti lisamine tekitab rea, mille vasakul pool on funktsiooni nimi ja selle muutujad ning paremal pool auk, mille asemele saab kirjutada funktsiooni definitsiooni. Enne, kui funktsiooni definitsiooni kirjutama saab hakata, tuleb inspekteerida funktsiooni argumente.

4.3.2. Juhtudeks jagamine

Juhtudeks jagamise käsk `:casesplit` jagab muutuja sellele vastavateks erinevateks väärtusteks või mustriteks. Kasutades eelnevas peatükis 4.3.1 saadud funktsioonis juhtudeks jagamise käsku muutuja `x` peal, jagab Idris selle automaatselt juhtudeks:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
zip_vect [] y = ?zip_vect_rhs_1
zip_vect (x :: xs) y = ?zip_vect_rhs_2
```

Muutujal `x` on kaks võimalust: kas see on vektor pikkusega null ehk tühi list või vektor pikkusega rohkem kui null ehk list, milles on elemente. Sarnaselt muutujale `x` tuleb nüüd inspekteerida ka muutujat `y`. Inspekteerides muutujat `y` teisel real, annab Idris vastuseks:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
zip_vect [] [] = ?zip_vect_rhs_1
zip_vect (x :: xs) y = ?zip_vect_rhs_2
```

Kuna esimese muutuja väärtus on tühi list, peab ka teine argument funktsiooni tüübi järgi tühi list olema. Inspekteerides muutujat `y` kolmandal real, asendab Idris selle vastavalt:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
zip_vect [] [] = ?zip_vect_rhs_1
zip_vect (x :: xs) (y :: ys) = ?zip_vect_rhs_2
```

Sarnaselt peab sõltuvalt muutujast `(x::xs)` ka muutuja `(y::ys)` olema elemente sisaldav list. See illustreerib, kuidas Idris suudab ühe sisendparameetri väärtuse järgi tuletada teise sisendparameetri väärtuse ja selle meie eest ise ära kirjutada.

Käsule `:casesplit` analoogiliselt töötab käsk `:addmissing`, mis lisab juba osaliselt defineeritud funktsioonile need sisendi juhud ja mustrid, mis on funktsiooni täielikkuse saavutamiseks veel puudu.

4.3.3. Augud ja otsimine

Auk tähistab programmi mittetäielikku osa ehk osa, mida pole veel kirjutatud [9]. Augud on süntaktiliselt lubatud, mis tähendab, et auke sisaldavad programmid lähevad tüübikontrollist läbi [9]. Küll aga juhib kompilaator aukudele tähelepanu, et neid hiljem defineerida ei unustataks. Auke tähistatakse süntaksiga `?auguNimi`.

Auku sobib ainult õige tüübiga väärtus. Seega saab tüübikontrollijalt küsida kontekstuaalset infot, mis on funktsiooni sisu kirjutamisel abiks. Selle osas, mida funktsioon tegema peaks, võimaldab esimest klassi tüüpide kasutamine olla aga väga täpne. Seega kui funktsiooni tüüp on täpselt antud, suudab tüübikontrollija vahel isegi mõned funktsiooni osad programmeerija eest ise ära täita.

Rakendades augul käsku `:proofsearch` püüab see leida augule sobiva väärtuse vastavalt selle tüübile. Otsides proovitakse väärtusteks lokaalseid muutujaid, rekursiivseid kutseid ja konstruktoreid [9]. Seejärel pakutakse neist sobivaim. Mida täpsem on tüübi kirjeldus, seda vähem on erinevaid võimalusi ja seda täpsem on välja pakutav lahendus.

Listifunktsioonide kirjutamisel ei ole otsimise käsust tihti abi, sest kui on võimalus,

et vastuseks sobib tühi list, siis see ka pakutakse. Küll aga töötab otsimine üpris hästi sõltuvate tüüpide, näiteks vektorite peal. Näiteks kui kahe vektori kokkupakkimise funktsioonist on defineeritud osa:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
zip_vect [] [] = ?zip_vect_rhs_1
zip_vect (x :: xs) (y :: ys) = ?zip_vect_rhs_2
```

Siis otsides võimalikke lahendusi aukudele `?zip_vect_rhs_1` ja `?zip_vect_rhs_2`, leiab Idris neile vastusteks:

```
zip_vect : Vect n a -> Vect n b -> Vect n (Pair a b)
zip_vect [] [] = []
zip_vect (x :: xs) (y :: ys) = (x, y) :: zip_vect xs ys
```

Brady rõhutab, et kuigi Idris suudab definitsioonidele õigeid vastuseid leida, peab ikkagi aru saama, mida tehakse. Kuna Idris võib eksida, tuleb üle kontrollida, kas pakutav vastus on päriselt see, mis arvatakse, et see olema peaks. Üldiselt aga kiirendab otsimine lihtsamate rekursioonide kirjutamist. Lisaks võib see abiks olla keerulisemate funktsioonide defineerimisel. Kui otsing suudab isegi pooleldi õige vastuse leida, on sellest õige vastuse tuletamine kohati kergem, kui selle tühjast välja mõtlemine [13].

4.4. Valemite tüüpidena väljendamine

Selle alampeatüki sisu baseerub suures osas F. Teegeni seminaritööl [14] ja Tallinna tehnikaülikooli funktsionaalprogrammeerimise kursuse 2020. aasta kevade 10. loengul, mida luges E. B. Morehouse [12].

Curry-Howard vastavus ütleb, et tüübid, tüübimuutujad ja termid on isomorfsed vastavalt lausearvutuse valemite, lausemuutujate ja tõestustega [14, 15]. Idrises tähendab see, et konstrueerides vastavat tüüpi avaldise, on sellega üheaegselt konstrueeritud ka tõestus, et nimetatud tüübile vastav valem kehtib [14]. Selle seose illustreerivaks näiteks on nii F. Teegeni seminaritöös kui ka Tallinna tehnikaülikooli funktsionaalprogrammeerimise kursusel defineeritud kaks predikaati: `Even` ja `Odd`, mis tähistavad vastavalt, kas tegu on paaris või paaritu arvuga [12, 14].

F. Teegen alustab paarsuse matemaatilise induktsiooniga. Esmalt defineeritakse kaks aksioomi. Üks aksioom eeldab, et null on paarisarv ning teine, et üks on paaritu arv. Seejärel defineeritakse induktiivselt reeglid:

1) Kui n on paarisarv, on ka $n + 2$ paarisarv.
2) Kui n on paaritu arv, on ka $n + 2$ paaritu arv.

Kirjeldatud aksioomid ja reeglid saab kirja panna vastavalt:

$$evenZero : Even(0) \qquad oddOne : Odd(1)$$
$$evenSucc : Even(n) \Rightarrow Even(n + 2) \qquad oddSucc : Odd(n) \Rightarrow Odd(n + 2)$$

Nende reeglite ja aksioomidega saab konstrueerida tõestuse ükskõik millisele paaris või paaritule naturaalarvule. Näiteks saab tõestada, et neli on paarisarv, kasutades aksioomil $evenZero$ kaks korda reeglit $evenSucc$. Sarnaselt saab tõestada, et viis on paaritu arv, rakendades reeglit $oddSucc$ kaks korda aksioomil $oddOne$ [14].

Morehouse defineeris sissejuhatuseks predikaadid kui tõeväärtust arvutavad funktsioonid. Seega defineeriti funktsioon `is_even`, mis arvutab, kas parameetrina antud naturaalarv on paarisarv või mitte, ning tagastab vastavalt `True` või `False`:

```
1 is_even : Nat -> Bool
2 is_even Z         = True
3 is_even (S Z)     = False
4 is_even (S (S n)) = is_even n
```

Seejärel argumenteeriti, et selline lähenemine pole piisav. Defineerides näiteks muutujad:

```
1 four_is_even : Bool
2 four_is_even = is_even 4
3
4 six_is_even : Bool
5 six_is_even = is_even 6
```

saab neid võrrelda valemis `four_is_even == six_is_even`. Kuna tõeväärtusel on ainult kaks valemit, tõene või väär, ei tule sellest välja erinevus, miks on neli või kuus paarisarvud. Samal põhimõttel saaks väita, et Fermat' suur teoreem ja `1+1 == 2` on sama väärtusega, mis ei ole aga täpne [12]. Kui täpsema konstruktsiooni vajalikkus on motiveeritud, defineeritakse predikaadid kui indekseeritud tüübid.

Kuivõrd valemid korreleeruvad tüüpidega, tuleb defineerida kaks tüübiperekonda Even ja Odd. Kuna paarsuse loogika on defineeritud naturaalarvudel, indekseeritakse nende valemite tüübikonstruktorid üle tüübi Nat. Konstruktoritena kasutatakse eelnevalt defineeritud aksioome ja induktiivseid reegleid [14].

```
data Even  : (n : Nat) -> Type where
  EvenZero : Even Z
  EvenSucc : Even n -> Even (S (S n))

data Odd  : (n : Nat) -> Type where
  OddOne  : Odd (S Z)
  OddSucc : Odd n -> Odd (S (S n))
```

Matemaatiliselt defineeritud reeglitele sarnaselt saab ka koodis tõestada, et arv neli on paaris ning arv viis on paaritu. Selleks tuleb eeldefineeritud konstruktorite abil konstrueerida väärtused tüüpidele Even 4 ja Odd 5:

```
fourIsEven : Even 4
fourIsEven = evenSucc (evenSucc evenZero)

fiveIsOdd : Odd 5
fiveIsOdd = oddSucc (oddSucc oddOne)
```

Idrise tüübikontrolli teostusega on tõestused verifitseeritud. On võimatu leida väärtust tüübiga Odd 4, mis tähendab, et see on tõestamatu. Teisisõnu, on väär, et neli on paaritu [14].

Kasutades eelnevalt defineeritud tõestuseid, saab nende kaudu edasi tõestada mitmeid paaris- ja paaritute arvude kohta käivaid väiteid. Näiteks saab tõestada väited:

1) kahe paarisarvu summa on paarisarv,
2) kahe paaritu arvu summa on paarisarv,
3) paaris ja paaritu arvu summa on paaritu arv,
4) kahe paarisarvu korrutis on paarisarv,
5) kahe paaritu arvu korrutis on paaritu arv,
6) paaritu ja paarisarvu korrutis on paarisarv.

Idrises saab neid implikatsioone väljendada funktsiooni tüübiga. Eeldused on funktsiooni parameetriteks ning järeldus funktsiooni tagastustüübiks. E. B. Morehouse tõi loengus

koodinäite 1. väite kohta:

```
even_plus_even : Even m -> Even n -> Even (m + n)
even_plus_even EvenZero     y = y
even_plus_even (EvenSucc x) y = EvenSucc (even_plus_even x y)
```

F. Teegeni seminaritöös on toodud aga koodinäide 2. väite kohta:

```
odd_plus_odd : Odd n -> Odd m -> Even (n + m)
odd_plus_odd OddOne OddOne       = EvenSucc EvenZero
odd_plus_odd OddOne (OddSucc y) = EvenSucc $ odd_plus_odd OddOne y
odd_plus_odd (OddSucc x) y       = EvenSucc $ odd_plus_odd x y
```

Kasutades funktsiooni `odd_plus_even` argumentidena eelnevalt defineeritud tõestust `fiveIsEven`, saab tõestada väite, et naturaalarv 10 on paarisarv:

```
tenIsEven : Even 10
tenIsEven = odd_plus_odd fiveIsEven fiveIsEven
```

Tallinna funktsionaalprogrammeerimise loengus tehti näitena läbi ka 4. väite tõestus. Korrutiste kohta käivate väidete tõestamise teeb aga keeruliseks asjaolu, et naturaalarvude korrutis on Idrises defineeritud rekursiivselt läbi liitmise:

```
mult : Nat -> Nat -> Nat
mult Z right        = Z
mult (S left) right = plus right $ mult left right
```

Tõestuse alustamine on võrdlemisi lihtne. Selleks defineeritakse funktsioon nimega `even_times_even`, kus on kaks põhijuhtu:

```
even_times_even : Even m -> Even n -> Even (m * n)
even_times_even EvenZero n_even            = EvenZero
even_times_even (EvenSucc m_even) n_even  = ?Goal_2
```

Esimese juhu parem pool on tuletatud tingimusest, et ükskõik millise naturaalarvu korrutamisel arvuga 0 on vastus 0. Teise juhu parema poole tuletamine aga nii lihtne ei ole. Augu `?Goal_2` tüüpi inspekteerides näitab Idris eesmärgina `Goal_2 : Even (plus n (plus n (mult n1 n)))`, kus kuvatakse muutujanimesid `n` ja `n1`, mida funktsiooni skoobis isegi justkui olemas ei ole. Lisaks sellele ei tea Idris ka seda, mis järjekorras käesolevat eesmärki tõestada tahetakse. Seetõttu ei saa sellisel viisil tõestust kirjutades auke inspekteerides eesmärkide kohta mõistlikku infot ning Idris

programmeerijale suureks abiks ei ole. Küll aga on olemas erinevad tehnikad, kuidas Idris end abistama suunata. Seda, millised need tehnikad täpselt on, saab vaadata TalTechi funktsionaalprogrammeerimise kursuse [12] 10. loengust. Järnevalt on toodud loengus tehtud tõestuse lõpplahendus:

```
even_times_even : Even m -> Even n -> Even (m * n)
even_times_even EvenZero n_even  =  EvenZero
even_times_even (EvenSucc m_even) n_even  =
  even_plus_even n_even (even_plus_even n_even (even_times_even m_even
    n_even))
```

4.5. Uute teemade käsitlemine Idrises

Neljanda peatüki põhjal vastati uurimisküsimuse teisele poolele: Milliste tänapäevaste teemade õpetamist Haskelli asendamine Idrisega võimaldab?

Idrise baasil saab tutvustada tugeva tüübisüsteemi ja sõltuvate tüüpidega programmeerimise eeliseid. Bakalaureuse astmes võib õpetada näiteks teemasid: tüüpidega arvutamine, sõltuvate tüüpidega funktsioonid, sõltuvad tüübid, interaktiivne programmeerimine ja valemite tüüpidena väljendamine. Verifitseerimise teemade käsitlemist Idrises saab edasi uurida B. C. Pierce jt. raamatu „Software Foundations“ E. Bailey jt. Idrise tõlkest [16].

5. Kokkuvõte

Selle bakalaureusetöö eesmärk oli analüüsida, kas sõltuvate tüüpidega programmeerimiskeel Idris on sobilik keel bakalaureuseastmes klassikalise ja tänapäevase funktsionaalprogrammeerimise õpetamiseks. Töö sisus analüüsiti, kuidas mõjutab Haskelli asendamine Idrisega „Programmeerimiskeelte“ kursusel käsitletud klassikaliste funktsionaalprogrammeerimise teemade käsitlemist ning milliste bakalaureuse tasemele sobivate tänapäevaste teemade käsitlemist asendus võimaldab.

Töö raames kohandati esmalt kursuse „Programmeerimiskeeled“ praktikumiülesanded Haskellist Idrisesse. Kohandamise tulemusena selgus, et keele vahetuse mõju klassikaliste funktsionaalprogrammeerimise teemade õpetamisele on väike. Küll aga tuleks kaaluda mõningate klassikalisi teemasid käsitlevate ülesannete asendamist. Seejärel anti ülevaade uutest teemadest, mille saab programmeerimiskeelele Idris üle minnes kursuse kavva võtta. Sealhulgas tutvustati sõltuvate tüüpidega programmeerimise eeliseid ning pakuti välja, kuidas teha sõltuvate tüüpide abil sissejuhatust verifitseerimisse.

Töö tulemusena sai väita, et Idris on sobilik keel bakalaureuseastmes klassikalise ja tänapäevase funktsionaalprogrammeerimise õpetamiseks.

Viidatud kirjandus

[1] Kalmer Apinis, Simmo Saan, Tartu Ülikooli kursus Progammeerimiskeeled, sügis 2020. https://courses.cs.ut.ee/2020/PK/fall/Main/Syllabus (10.12.2020)

[2] Simon Thompson. Haskell: The Craft of Functional Programming. Second Edition. Harolow: Pearson Education Limited. 1999.

[3] Erik Meijer, Delft University of Technology OpenCourseWare MOOC Introduction to Functional Programming, 2014. https://ocw.tudelft.nl/courses/introduction-to-functional-programming (09.01.2021)

[4] Härmel Nestra. Sissejuhatus funktsionaalsesse programmeerimisse. Tartu: Tartu Ülikooli Kirjastus. 2010.

[5] Idris Documentation. https://www.idris-lang.org/ (12.12.2020)

[6] Edwin Brady, Idris 2: Quantitative Type Theory in Action, 2018. https://www.type-driven.org.uk/edwinb/papers/idris2.pdf (12.12.2020)

[7] Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler, A History of Haskell: Being Lazy With Class, 2007. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf (25.04.2021)

[8] Glasgow Haskell Compiler. https://gitlab.haskell.org/ghc/ghc/-/wikis/dependent-haskell (29.04.2021)

[9] Edwin Brady. Type-Driven Development with Idris. New York: Manning Publications. 2017.

[10] Cliff Harvey, Lightning Intro to Functional Programming and Dependent Types with Idris. https://fieldstrength.org/learn-idris/ (03.03.2021)

[11] David Christiansen, Derive all the instances, 2018. https://github.com/david-christiansen/derive-all-the-instances (28.04.2021)

[12] TalTech Functional Programming course, spring 2020. https://compose.ioc.ee/CourseFunctionalProgramming.html (11.02.2021)

[13] Edwin Brady, Type-Driven Development in Idris — Edwin Brady, Scala World 2015. https://www.youtube.com/watch?v=X36ye-1x_HQ&t=868s&ab_channel=ScalaWorld (18.03.2021)

[14] Finn Teegen, Idris: A Functional Programming Language with Dependent Types, 2015. https://www-ps.informatik.uni-kiel.de/~mh/lehre/seminare/ws14_master/docs/Teegen.pdf (22.04.2021)

[15] Varmo Vene, Funktsionaalne programmeerimine, kevad 2007. http://kodu.ut.ee/~varmo/FP2007/ (14.04.2021)

[16] Benjamin C. Pierce and Others, Software Foundations, 2011, Idris translation by Eric Bailey, Alex Gryzlov and Erlend Hamberg. https://idris-hackers.github.io/software-foundations/pdf/sf-idris-2018.pdf (18.04.2021)

Lisad

I Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Karoliine Holter**,

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

   Funktsionaalprogrammeerimise õpetamine,

   mille juhendajad on Kalmer Apinis ja Vesal Vojdani,

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

2) Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3) Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4) Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Karoliine Holter
07.05.2021