Matias Keveri

Opiskelijoiden suurimmat haasteet Haskell-ohjelmointikielen tyyppijärjestelmän kanssa.

Tietotekniikan pro gradu -tutkielma

22. huhtikuuta 2018

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Matias Keveri

Yhteystiedot: maankeve@jyu.fi

Ohjaaja: Ville Tirronen

Työn nimi: Opiskelijoiden suurimmat haasteet Haskell-ohjelmointikielen tyyppijärjestelmän kanssa.

Title in English: Main difficulties students have with Haskell's type system.

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 69+1

Tiivistelmä: Haskell-ohjelmointikielellä opetettavalla funktio-ohjelmoinnin johdatuskurssilla oppilaat kohtaavat useita haasteita. Näistä yleisimmät liittyvät usein kielen syntaksiin tai tyyppijärjestelmään. Tämä tutkielma keskittyy oppilaiden haasteisiin Haskellin tyyppijärjestelmän kanssa laajentaen aiempaa tutkimusta tyyppeihin liittyvistä haasteista. Tavoitteena on tunnistaa oppilaiden yleisimmät virheet, haasteet ja väärinkäsitykset tyyppeihin liittyen analysoimalla automaattitehtävistä kerättyä aineistoa. Analyysi keskittyy tyyppien ymmärrystä testaaviin tehtäviin, joista saatavien havaintojen pohjalta opetusta voidaan keskittää haasteellisimpiin tyyppijärjestelmän osa-alueisiin.

Avainsanat: funktio-ohjelmointi, tyyppijärjestelmät, oppiminen

Abstract: Students face many challenges when attending an introductory functional programming course taught in Haskell. Common challenges are often related to syntax of the language or the type system. This paper focuses on the difficulties of students in understanding Haskell's type system extending the current research on type related difficulties. Our goal is to identify common mistakes, difficulties and misconceptions students have with types by analyzing exercise submissions and automated assignment logs. The analysis will be focused on assignments testing the students knowledge on types. After identifying the main difficulties, the courses teaching can be more focused on the problematic parts.

Keywords: Programming education, Functional programming, Type systems, Haskell, Beginner difficulties, Function types

Kuviot

Taulukot

Sisältö

1 Johdanto

Funktio-ohjelmointikielten on ehdotettu tarjoavan monia hyötyjä niin aloittelijoille kuin kokeneemmillekin ohjelmoijille opetettaessa johdatuskurssia ohjelmointiin (Felleisen ym. 2004). Iso osa useiden funktio-ohjelmointikielten hyödyistä tulee niiden täsmällisistä tyyppijärjestelmistä, jotka mahdollistavat tarkan formaalin ohjelmien tarkastuksen (Cardelli 1996). Tämän avulla ohjelmoijat voivat esittää aikeensa tarkasti ja saada paremman varmuuden ohjelman toimimisesta. Huolimatta tyyppijärjestelmien hyödyllisyydestä, on niiden raportoitu olevan suuria lähteitä aloittelevien ohjelmoijien ongelmille (Joosten, Van Den Berg ja Van Der Hoeven 1993; Clack ja Myers 1995).

Tässä tutkielmassa tarkastellaan funktio-ohjelmoinnin kurssille osallistuneiden opiskelijoiden haasteita Haskell-ohjelmointikielen tyyppijärjestelmän kanssa. Kyseisellä kurssilla on aiemmin tutkittu vaikeuksia ja väärinkäsityksiä modernin tyyppijärjestelmän kanssa (Tirronen 2014), sekä aloittelijoiden virheitä Haskell-ohjelmointikieleen liittyen (Tirronen, Uusi-Mäkelä ja Isomöttönen 2015). Tämä tutkielma laajentaa tutkimusta aloittelijoiden haasteisiin funktio-ohjelmoinnissa ja tarkemmin Haskell-ohjelmointikielessä keskittymällä kielen tyyppijärjestelmään.

Tavoitteena on tunnistaa opiskelijoiden yleisimmät virheet, väärinymmärrykset ja haasteet tyyppijärjestelmän kanssa analysoimalla tehtävien palautuksia ja automaattitehtävien keräämiä lokitiedostoja. Kurssilla aineistoa kerätään tallentamalla kaikki tehtävien palautukset ja niitä voidaan tarkastella jälkeenpäin. Osana tutkielmaa kurssille tehdään myös lisää automaattitehtäviä, jotta saadaan kerättyä enemmän aineistoa liittyen tyyppijärjestelmään. Kerättyä aineistoa analysoidaan automaattisesti etsien yleisiä ongelmia liittyen tyyppijärjestelmään. Tulosten perusteella kurssin opetusta voidaan keskittää tyyppijärjestelmän vaikeiksi havaittuihin kohtiin.

Tutkielma rakentuu seuraavasti. Toisessa luvussa käsitellään tyyppijärjestelmiä ja niiden hyötyjä. Kolmannessa luvussa tutustutaan Haskell-ohjelmointikieleen, sekä sen tyyppijärjestelmään. Neljännessä luvussa tutustutaan aiheeseen liittyvään tutkimukseen, etenkin Haskellia ja tyyppijärjestelmiä koskevaan. Viidennessä luvussa käsitellään kurssia yleises-

ti ja sen tehtäviä. Kuudennessa luvussa esitellään tutkimuskysymykset, tutkimusmenetelmät ja aineiston keruu. Seitsemännessä luvussa käsitellään aineiston analysointia ja tutkimuksen rajoitteita. Kahdeksannessa luvussa esitellään tutkimuksen tulokset ja jatkotutkimusmahdollisuudet.

2 Tyyppijärjestelmät

Tyyppijärjestelmät ovat erittäin olennainen osa ohjelmointikieliä ja vaikuttavat erittäin paljon kielten ominaisuuksiin.

Tässä luvussa käsitellään tyyppijärjestelmiä. Aluksi niiden määritelmää ja käsitettä yleisemmin. Sitten hieman staattisen ja dynaamisen tyypityksen eroista. Lopuksi vielä käydään läpi tyyppijärjestelmien yleisiä hyötyjä. Yksityiskohtaisempiin tyyppijärjestelmien asioihin mennään seuraavassa luvussa, jossa keskitytään Haskell-ohjelmointikielen tyyppijärjestelmän ominaisuuksiin.

2.1 Määritelmä

Tyyppijärjestelmän keskeinen tarkoitus on estää mahdolliset ohjelman suoritusaikana esiintyvät virheet. Ne tarjoavat myös keinon järkeillä ja ymmärtää ohjelmia. Pierce 2002 huomauttaa tyyppijärjestelmän käsitteen olevan haastava määriteltävä siten, että se kattaa ohjelmointikielten suunnittelijat ja toteuttajat ollen silti jokseenkin merkittävä. Hän myös antaa seuraavan määritelmän tyyppijärjestelmälle: Tyyppijärjestelmä on mukautuva syntaktinen metodi tietynlaisten ohjelman käytösten poissaolon todistamiseen luokittelemalla lausekkeet sen mukaan millaisiksi arvoiksi ne evaluoidaan.

Yleisemmällä tasolla termi tyyppijärjestelmä viittaa laajempaan logiikan, matematiikan ja filosofian tutkimusalueeseen. Tyyppijärjestelmät siinä merkityksessään formalisoitiin ensimmäistä kertaa 1900-luvun alkupuolella keinoiksi vältellä loogisia paradokseja, jotka uhkasivat matematiikan perusteita. 1900-luvun aikana tyypeistä on tullut standardityökaluja logiikan ja etenkin todistusteorian parissa. Ne ovat myös saapuneet filosofian ja tieteen kieleen. (Pierce 2002) Alonso Church oli loogikko ja matemaatikko, joka julkisti lambdakalkyylin 1930-luvulla ja myöhemmin tutki sen pohjalta tyyppiteoriaa (Church 1940).

Tyyppijärjestelmät siis auttavat käyttäjiä tuomalla rakennetta ohjelmiin erilaisten tyyppien muodossa. Ne tuovat selkeämpiä abstraktioita konekielen päälle. Tällöin käyttäjän ei

tarvitse miettiä yksittäisiä tavuja ja niiden sijaintia ja järjestystä muistissa, vaan käyttäjä voi ohjelmoida tyyppien tasolla käyttäen esimerkiksi liukulukuja ja merkkijonoja. Nämä tyypit ovat vielä erittäin yksinkertaisia ja tyyppijärjestelmät tuovatkin mahdollisuuden käyttää ja määritellä monipuolisesti erilaisia tyyppejä. Tämä auttaa abstraktioiden luomisessa ja ohjelmakoodin järjestelyssä tuoden jonkinlaisen rakenteen ja tyyppitarkastuksen mahdollisuuden ohjelmaan.

2.2 Tyyppitarkastus

Tyyppijärjestelmän tyyppien rajoitusta, varmistusta ja tarkastusta kutsutaan tyyppitarkastukseksi, joka voi tapahtua käännöksen aikana tai ajonaikana. Käännöksen aikana tapahtuvaa tyyppitarkastusta kutsutaan staattiseksi tyyppitarkastukseksi, kun taas ajonaikana tapahtuvaa dynaamiseksi tyyppitarkastukseksi. Joskus käytetään myös ilmaisua “dynaamisesti tyypitetty”, joka ei ole täsmällinen, vaan osuvampi termi olisi “dynaamisesti tarkastettu”. Harper 2016 mainitsee, että dynaamisten kielten usein ajatellaan olevan staattisten kielten vastakohta, mutta tämä vastakkainasettelu on illusorinen. Hänestä dynaamisten kielten voidaan ajatella olevan staattisten kielten erikoistapauksia, joissa on käytössä vain yksi tyyppi. Tällöin staattiset ja dynaamiset tyypit olisivat kokonaan eri kategorioissa, mutta näiden eroa ei ole olennaista tutkia syvemmin tämän tutkimuksen kannalta.

Tyypityksen avulla annetaan merkitys tietokoneen muistissa olevalle tiedolle. Tieto on muistissa peräkkäisinä tavuina ja tietokone ei osaa erottaa tavuista merkkijonoa tai liukulukua ilman tyypityksen tukea. Ohjelmointikielissä muistissa olevalla tiedolla on aina tyyppi, mutta sen merkityksellisyys voidaan tarkistaa käännöksen aikana staattisesti tai vasta suorituksen aikana dynaamisesti.

Vaikka staattisesti tyypitetyt ohjelmat onkin tyyppitarkastettu käännöksen aikana hyödyntävät ne silti myös ajonaikaisia tarkastuksia muiden kuin tyyppivirheiden varalta. Esimerkiksi yleinen virhe, jossa ohjelmoija osoittaa indeksillä listan ulkopuolelle on tarkastettava ajonaikana. Ohjelmointikieli voi rajoittaa tämän tyyppisten virheiden mahdollisuutta tarjoamalla rakenteita, jotka mahdollistavat saman toiminnallisuuden ilman suoria viittauksia indeksillä listaan. Tyyppien kannalta näissä virhetilanteissa kaikki on oikein, joten näitä ei tule sekoittaa

tyyppivirheisiin. Ajonaikaisten virheiden tarkastukseen viitataan joskus dynaamisella tarkastuksella, mutta tätä ei tule sekoittaa dynaamiseen tyyppitarkastukseen, joka tarkastaa vain osan mahdollisista ajonaikaisista virheistä.

Jotkut dynaamista tyyppitarkastusta käyttävät kielet hyödyntävät staattista tyyppitarkastusta tarjoamalla käyttäjälle mahdollisuuden lisätä tyyppejä lähdekoodiin. Tällaisen mahdollisuuden tarjoaa esimerkiksi Clojure-ohjelmointikieli, joka ei vaadi tyyppejä lähdekoodiin, mutta niitä voidaan lisätä tukemaan dokumentaatiota ja kääntäjää lisäosan muodossa.

2.2.1 Staattinen tyyppitarkastus

Staattisen tyyppitarkastuksen pääidea on analysoida ohjelmaa sen lähdekoodin perusteella. Lähdekoodista pyritään tyyppien avulla löytämään virheitä kääntäjän suorittaman tyyppitarkastuksen avulla. Staattista tyyppitarkastusta käytetään useissa ohjelmointikielissä, kuten Java, C, C++, C#, Scala, Haskell. Staattisen tyyppitarkastuksen suurimpia etuja on sen mahdollisuus ilmoittaa tietyistä virheistä käyttäjälle jo ohjelman kääntämisvaiheessa. Sen voidaankin ajatella olevan rajoittunut versio ohjelmien verifioinnista, koska se voi todistaa tietynlaisten virheiden poissaolon. Se minkä kaikkien virheiden poissaolon se pystyy todistamaan riippuu paljon kielestä ja kielen tyyppijärjestelmän toteutuksesta. Tyyppijärjestelmien hyötyjä käydään läpi seuraavassa kappaleessa tarkemmin.

Tyyppijärjestelmien ollessa staattisia ne välttämättömästi ovat myös konservatiivisia. Tarkoittaen, että ne voivat kategorisesti todistaa joidenkin epätoivottujen toiminnallisuuksien poissaolon ohjelmassa, mutta eivät voi todistaa niiden läsnäoloa ja siten hylkäävät joskus ohjelmia, jotka käyttäytyisivät oikein ajonaikana. Esimerkiksi alla esitetty yksinkertainen ohjelma 2.1 hylättäisiin huonosti tyypitettynä, vaikka “<complex test>” -osa evaluoituisikin aina todeksi, koska staattinen analyysi ei voi määrittää näin aina tapahtuvan. Jännite täsmällisyyden ja ilmaisuvoimaisuuden välillä on keskeinen fakta tyyppijärjestelmien suunnittelussa. Halu mahdollistaa useampien ohjelmien tyypitys tarkempia tyyppejä käyttäen on pääasiallinen voima, joka ajaa tutkimusta tieteenallalla. (Pierce 2002)

Listing 2.1. Väärin tyypitetty

```
if <complex test> then 5 else <type error>
```

Yksinkertaisessa ohjelmassa 2.1 käytetty ilmaisu “<type error>” tarkoittaa yleistä tyyppivirhettä. Tässä esimerkissä aiheutuisi siis tyyppivirhe, jos ehtolauseessa päädyttäisiin “else”-osioon.

2.2.2 Dynaaminen tyyppitarkastus

Tunnettuja dynaamisestityypitettyjä kieliä ovat esimerkiksi Python, Ruby ja useat LISP-kielet kuten Clojure. Dynaamiseen tyyppitarkastukseen luottavat kielet mahdollistavat tiiviimmän syntaksin ja lähdekoodi voi joidenkin mielestä olla luettavampaa ja selkeämpää, koska se sisältää vain oleellisen. Vaihtokauppana pelkkään dynaamiseen tyyppitarkastukseen luottavat kielet ovat alttiimpia ajonaikana ilmeneville virheille, koska niitä ei ennakkoon käännettäessä voi tarkastaa.

Dynaamista tyyppitarkastusta käytetään myös staattisen tyyppitarkastuksen tukena useissa kielissä esimerkiksi aiemmin mainitun listan käsittelyssä indekseillä ja myös tyyppimuunnoksissa (casting). Tyyppimuunnoksissa jotain tietoa pyritään muuntamaan tietyksi tyypiksi ajonaikana ja tätä ei usein voida staattisesti tarkastaa, joten dynaaminen tyyppitarkastus ajonaikana tukee tyyppijärjestelmää.

Dynaamista tyyppitarkastusta ei kuvailla tarkemmin tässä tutkielmassa, koska se ei ole oleellinen osa tutkielman kannalta.

2.3 Hyödyt

Tässä kappaleessa käydään läpi tyyppijärjestelmien tuomia hyötyjä. Tyyppijärjestelmät tukevat ohjelmistokehitysprosessia etenkin abstraktioiden, dokumentaation ja virheiden havaitsemisen kautta. Ne myös takaavat joillain kielillä kielen turvallisuutta, jolloin käyttäjä voi luottavaisemmin käyttää kieltä. Tyyppijärjestelmät myös lisäävät kääntäjän mahdollisuuksia tehostaa ja optimoida ohjelman käännettyä versiota.

2.3.1 Virheiden havaitseminen

Tyyppijärjestelmät auttavat virheiden löytämisessä. Pierce 2002 Kuvailee virheiden löytämistä seuraavan kappaleen verran. Selkein staattisen tyyppijärjestelmän hyöty on sen mahdollisuus havaita aikaisin joitain ohjelmointivirheitä. Ajoissa havaitut virheet voidaan korjata heti sen sijaan, että ne löydettäisiin paljon myöhemmin, kun ohjelmoijalla on jo keskellä seuraavaa asiaa tai vasta silloin kun ohjelma on jo tuotannossa. Erityisesti virheet voidaan usein löytää tarkemmin tyyppitarkistuksen aika, kuin ajonaikana jossa niiden vaikutuksien esiintulo voi viedä aikaa. Käytännössä staattinen tyyppitarkastus paljastaa yllättävän laajan määrän erilaisia virheitä. Rikkaasti tyypitettyjen kielten parissa työskentelevät ohjelmoijat usein kuvaavat ohjelmien "vain toimivan", kun ne ovat läpäisseet tyyppitarkistuksen. Paljon useammin kuin he itse uskoisivat.

Staattisen tyyppitarkastuksen yksi suurimpia hyötyjä on siis virheiden havaitseminen mahdollisimman aikaisin. Tämä usein nopeuttaa ohjelmistokehitystä ja vähentää tietynlaisten testien tarvetta. Kleinschmager ym. 2012 havaitsivat staattisen tyyppijärjestelmän huomattavasti vähentävän ohjelmointitehtävien tyyppivirheisiin kuluvaa kehitysaikaa. Myös Hanenberg ym. 2014 huomasivat staattisen tyyppijärjestelmän parantavan muun muassa tyyppivirheiden aiheuttamien virheiden korjaamista.

Ilman staattisen tyyppitarkastuksen tuomaa turvaa käyttäjä usein joutuu tekemään testejä myös yksinkertaisille tyyppeihin liittyville tilanteille, jotta voi saada paremman varmuudentunteen ohjelman toimivuudesta. Tyyppijärjestelmät eivät kuitenkaan automaattisesti tuo kaikkia hyötyjä suoraan käyttäjälle vaan käyttäjältä vaaditaan myös kielen tyyppijärjestelmän ymmärrystä, jotta hän voi ohjata tyyppijärjestelmää oikeilla abstraktioilla. Tästä seuraavassa kappaleessa Piercen ajatuksia.

Maksimaalisen hyödyn saamiseksi tyyppijärjestelmästä tarvitaan usein huomioita ohjelmoijalta, kuten myös halukkuutta hyödyntää kielen tarjoamia mahdollisuuksia. Esimerkkinä monimutkainen ohjelma, joka ilmaisee kaikki tietorakenteet listoina ei saa kaikkea mahdollista hyötyä ja apua kääntäjältä, toisinkuin ohjelma, joka määrittelee kaikille tietorakenteille omat abstraktit tietotyypit. Ilmaisuvoimaiset tyyppijärjestelmät tarjoavat lukuisia "temppuja" rakenteen koodaamiseen tyyppien muodossa. (Pierce 2002)

Tyyppijärjestelmä usein tukee myös ohjelman ylläpitoa ja refaktorointi prosessia. Kun käyttäjä muuttaa yhtä tyyppiä tai yhteen tyyppiin liittyviä ominaisuuksia voi kääntäjä ilmoittaa heti missä kaikkialla kyseinen muutos aiheuttaa ongelmia. Käyttäjä voi täten korjata muutoksen aiheuttamat virheet heti sen sijaan, että nämä virheet ilmenisivät myöhemmin ajonaikana. Staattinen tyyppitarkastus siis tukee hyvin ohjelmien ylläpitoa ja muokkausta.

2.3.2 Abstraktiot

Tyyppijärjestelmät auttavat ja tukevat erilaisia abstraktioita ja mahdollistavat asioiden esittämisen ja käsittelyn korkeammalla tasolla, jolloin ratkaisut ovat helpompia muokata ja ymmärtää verrattuna matalamman tason toteutuksiin, joissa ohjelmoijan täytyy kuvata asiat erityisen tarkasti ja runsassanaisesti. Pierce 2002 kuvaa abstraktioiden hyötyjä seuraavasti. Sen lisäksi ison skaalan ohjelmistojen kontekstissa tyyppijärjestelmät muodostavat selkärangan "moduulikielille", joiden avulla paketoidaan ja yhdistetään laajojen järjestelmien eri komponentit. Tyypit esiintyvät moduulien rajapinnoissa ja voidaan myös ajatella rajapintojen olevan moduulien tyyppejä tarjoten tiivistelmän moduulin ominaisuuksista. Tämän voidaankin ajatella olevan eräänlainen sopimus moduulin toteuttajan ja käyttäjän välillä.

Laajojen järjestelmien järjestäminen moduuleilla, joissa on selkeät rajapinnat johtaa abstraktimpaan tyyliin suunnittelussa, jossa rajapinnat on suunniteltu ja niistä on keskusteltu erillään lopullisesta toteutuksesta. Abstraktimpi rajapintojen ajattelu johtaa yleensä parempaan suunnitteluun. (Pierce 2002)

Tyyppijärjestelmät tukevat abstraktioita, jotka taas johtavat parempaan suunnitteluun ja siten selkeämpiin ja helommin ylläpidettäviin sovelluksiin.

2.3.3 Dokumentaatio

Tyyppijärjestelmät tukevat ohjelmien dokumentointia. Pierce 2002 kirjoittaa tyypeistä dokumentaationa seuraavaa. Tyypit ovat myös hyödyllisiä ohjelmia luettaessa. Tyyppimääritelmät proseduurien otsikoissa ja moduulien rajapinnoissa muodostavat tietynlaisen dokumentaation, joka antaa käyttäjälle vihjeitä käyttäytymisestä. Sen lisäksi tämän tyyppinen dokumentaatio ei voi vanheta, koska se on tarkistettu joka kerta ohjelmaa käännettäessä kääntäjän

toimesta, toisinkuin kommentteihin upotetut toiminnallisuuden kuvaukset. Tämä tyyppien rooli on tärkeä etenkin moduulien kuvauksissa.

2.3.4 Kielen turvallisuus

Tyyppijärjestelmien yhteydessä esiintyy usein myös käsite “turvallinen kieli”. Tällaista ilmaisua käytetään usein osaamatta välttämättä tarkemmin kuvailla mikä sen aiheuttaa. Staattinen tyypitys ei ainoastaan riitä tuomaan tätä tunnetta kielen käyttäjälle. Turvallisuuden tunne tulee kielillä, jotka suojaavat käyttäjää omilta abstraktioiltaan. Esimerkiksi turvalliseksi luokiteltavissa kielissä kuten Java tai Haskell käyttäjä ei voi lista-tietorakennetta päivittäessään kirjoittaa yli kyseisen listan muistialueen. Eli näiden kielien abstraktio listasta tuntuu turvalliselta. Tilanne on toinen kielissä kuten C tai C++, joissa käyttäjä on suuremmassa vastuussa ja voi listaan kirjoittaessaan kirjoittaa myös sen ylikirjoittaa seuraaviin muistialueisiin ja täten aiheuttaa suuria ongelmia.

Kielen turvallisuus ei ole kuitenkaan sama asia kuin staattinen tyypitys. Kielen turvallisuus voidaan saavuttaa staattisella tarkistuksella, mutta myös ajonaikana tehtävillä tarkistuksilla, jotka ottavat kiinni järjettömät operaatiot juuri sillä hetkellä kun niitä yritetään ja joko pysäyttävät ohjelman tai nostavat poikkeuksen. Esimerkiksi Scheme on turvallinen kieli, vaikka siinä ei ole staattista tyyppijärjestelmää. (Pierce 2002)

2.3.5 Tehokkuus

Tyyppijärjestelmät auttavat myös optimoinnissa ja tuovat siten tehokkuutta kieliin. Kääntäjä voi tyyppien perusteella päätellä miten ja millaisissa osissa tieto tulee olemaan muistissa ja siten käyttää mahdollisimman optimaalisia käskyjä tiedon käsittelyyn. Myös käskyjä voidaan poistaa, jos kääntäjä pystyy turvallisissa kielissä päättelemään tyyppien perusteella joidenkin operaatioiden varmuuden.

Tyyppien tietoihin perustuvat tehokkuuden parannukset voivat tulla yllättävistä paikoista. Esimerkiksi viime aikoina on osoitettu ohjelmakoodin generoinnin lisäksi myös osoitinten esityksiä voidaan parantaa tyyppianalyysin informaation perusteella rinnakkaisissa tieteellisissä ohjelmissa. (Pierce 2002)

3 Haskell ohjelmointikieli

Haskell on staattisesti tyypitetty puhdas funktio-ohjelmointikieli, joka eroaa huomattavasti yleisemmistä imperatiivisista ohjelmointikielistä. Funktiotyylin korkean tason luonne johtaa siihen, että Haskellilla kirjoitetut ohjelmat ovat usein paljon ytimekkäämpiä kuin muissa kielissä (Hutton 2007). Tässä luvussa tutustutaan Haskell ohjelmointikieleen lyhyesti. Ensin käydään läpi Haskellin ominaispiirteitä ja lopuksi tutustutaan kielen tyyppijärjestelmään sen verran mitä tässä tutkielmassa tarvitsee.

3.1 Ominaisuudet

Haskellin ollessa funktio-ohjelmointikieli esiintyy siinä useita yleisiä ominaispiirteitä funktio-ohjelmoinnista. Näitä piirteitä ovat esimerkiksi korkeamman tason funktiot, muuttumattomat tietorakenteet ja rekursion käyttö iteroinnissa. Näiden seurauksena funktiokielissä usein voidaan määritellä ongelmat lausekkeina, joiden avulla saadaan evaluoinnilla ratkaisut. Harvemmin määritellään tarkasti yksityiskohtaisia askelia tuloksen saavuttamiseen vaan määritelläänkin ongelma niin sanotusti korkeammalla tasolla. Keskitytään siis enemmän siihen mitä halutaan saavuttaa eikä miten se askel askeleelta saavutetaan.

Kappaleessa kuvataan Haskellin syntaksia esimerkein ja tuodaan esille tärkeitä funktio-ohjelmoinnin piirteitä kielestä. Nämä ominaisuudet vaikuttavat siihen, kuinka tyyppijärjestelmää käytetään ja kuinka selkeältä se tuntuu.

3.1.1 Syntaksi

Haskellin syntaksi on tiivistä ja etenkin jos muuttujat on nimetty lyhyesti tai pelkillä kirjaimilla voi syntaksi olla jopa kryptistä. Järkevällä muuttujien nimeämisellä ohjelmat pysyvät hyvin luettavina myös kieltä vähemmän tuntevalle. Syntaksi mahdollistaa usein algoritmien ja muiden halutuiden ratkaisujen ilmaisun hyvin selkeästi, kuten alla olevista esimerkeistä (3.1, 3.2) nähdään.

Seuraava esimerkki 3.1 kuvastaa Haskellin syntaksin tiiviyttä ja olennaiseen keskittymistä. Alla on esitetty Fibonaccin lukujono matemaattinen määritelmä rekursiivisesti. Fibonaccin lukujonossa seuraava luku lasketaan aina summaamalla kaksi edellistä lukua.

$$F(n) = \begin{cases} 0 & ,\text{kun } n = 0, \\ 1 & ,\text{kun } n = 1 \\ F(n-1) + F(n-2) & ,\text{kun } n > 1 \end{cases}$$

Lukujono on toteutettu Haskellilla koodiesimerkissä 3.1. Näitä vertaamalla on selkeää, että Haskellin syntaksi muistuttaa matematiikkaa melko paljon.

Listing 3.1. Fibonaccin lukujono Haskell funktiona

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Toinen yleinen esimerkki kielen selkeydestä on pikalajittelu (quicksort) algoritmi on yleinen esimerkki kun esitellään Haskell ohjelmointikieltä. Esimerkki 3.2 kuvastaa hyvin kielen korkean tason luonnetta ongelmien kuvaamiseen ja ratkaisemiseen.

Listing 3.2. Pikalajittelu algoritmi Haskell funktiona

```
pikalajittelu []     = []
pikalajittelu (x:xs) = pienet ++ [x] ++ suuret
  where
    pienet = pikalajittelu [a | a <- xs, a <= x]
    suuret = pikalajittelu [a | a <- xs, a > x]
```

3.1.2 Funktio-ohjelmoinnin piirteet

Funktio-ohjelmoinnissa yksi tärkeä konsepti on korkeamman asteen funktiot (higher-order functions). Haskell on korkeamman asteen funktiokieli, joka tarkoittaa sitä että funktiot voivat vapaasti ottaa muita funktioita argumentteinaan ja palauttaa tuloksena funktioita

(Hutton 2007). Tämä ominaisuus mahdollistaa uusien funktioiden muodostamisen yhdistämällä muita funktioita. Haskellissa funktiot ovat myös oletuksena puhtaita, eli niillä ei ole sivuvaikutuksia. Tällöin voidaan ajatella funktion kuvaavan lähtöjoukon kaikki alkiot joka kerta samalla tavalla. Funktio tuottaa siis saman arvon samalle syötteelle riippumatta ajasta ja paikasta. Funktioista joilla ei ole sivuvaikutuksia käytetään myös termiä viittauksiltaan läpinäkyvä (referentially transparent). Ohjelmia on erittäin helppo järkeillä puhtaiden funktioiden osalta, koska niitä voi ajatella matemaattisemmin.

Muuttumattomat (immutable) tietorakenteet ovat myös hyvin yleisiä funktiokielissä ja Haskell:kin käyttää suurimmaksi osaksi vain tämänkaltaisia tietorakenteita. Tiedon muuttumista on helppo seurata näin, koska siihen voi vaikuttaa vain funktio, jossa sitä käsitellään. Tietorakenteita muuttaessa luodaan siitä uusi versio kopioimalla ja alkuperäinen säilyy muuttumattomana. Tämä voi aluksi vaikuttaa tehottomalta, mutta ei sitä kuitenkaan ole, koska alkuperäinen tietorakenne säilyy varmasti muuttumattomana voidaan sen osia jakaa uusien tietorankenteiden kanssa.

Muutokset ovat suurimmassa osassa ohjelmia kuitenkin välttämättömiä. Haskellin tapa toteuttaa funktiot jotka eivät ole viittauksiltaan läpinäkyviä on hyödyntää matematiikasta tuttua käsitettä monadit. Monadeja ei tässä vaiheessa tarvitse määritellä tarkemmin vaan voidaan ajatella niiden olevan tapa ilmaista mahdollisia sivuvaikutuksia. Tärkeintä on huomioida se, että Haskellissa täytyy syntaktisesti ilmaista paikat, joissa sivuvaikutukset ovat mahdollisia ja näin saadaan kääntäjän avulla varmistettua, että puhtaat funktiot pysyvät varmasti puhtaina.

Haskell ohjelmia suoritetaan käyttäen tekniikkaa nimeltä “laiska evaluaatio” (lazy evaluation), joka pohjautuu siihen ideaan, että mitään laskentaa ei tulisi suorittaa ennen kuin siihen liittyvää tulosta tarvitaan (Hutton 2007). Koska puhtaat funktiot ovat viittauksiltaan läpinäkyviä, voidaan ne suorittaa milloin tahansa ja ne palauttavat aina saman tuloksen. Laiska evaluaatio mahdollistaa esimerkiksi äärettömät tietorakenteet.

Iterointi saavutetaan funktiokielissä yleensä rekursion avulla, sillä se on sääntöjen mukainen ja usein ainut tapa iteroida. Rekursiivisten funktioiden ideana on kutsua itseään niin kauan kunnes saavutetaan lopetusehto (base case). Rekursion käyttö ei myöskään kasvata pinoa

(stack), koska käytetään häntäkutsun optimointia (tail call optimization).

3.2 Tyyppijärjestelmä

Haskellissa on staattinen tyyppitarkastus ja sen tyyppijärjestelmä on erittäin monipuolinen verrattuna moniin muihin ohjelmointikieliin. Tässä kappaleessa kuvataan Haskellin tyyppijärjestelmää yleisesti keskittyen asioihin ja konsepteihin, joita kieltä oppivat opiskelijat alussa kohtaavat. Tämä toimii pintaraapaisuna kielen tyyppijärjestelmään, mutta kaikki oleellinen tutkielmaan liittyvä taustatieto löytyy. Kieltä opiskelevat oppilaat eivät vielä ensimmäisellä kurssilla ehdi kovin monipuolisiin tehtäviin ja siten riittää käydä yksinkertaisimmat ominaisuudet läpi. Näitä ominaisuuksia ovat esimerkiksi tyyppien päättely (type inference), tyypit, funktiotyypit, tyyppiluokat sekä algebralliset tietotyypit (algebraic data types).

3.2.1 Tyyppien päättely

Staattisen tyyppitarkastuksen vuoksi Haskellia kirjoitetaan tiedostoon tyyppinotaatioiden kera ja sitten ohjelmakoodi tarkastetaan kääntäjän avulla. Kääntäjä kertoo virheilmoitusten avulle käyttäjälle, jos se ei voi päätellä annettujen tyyppien avulla toimivaa ohjelmaa. Kaikkialle ei kuitenkaan tarvitse kirjoittaa tyyppinotaatiolla tyyppejä, koska tyyppijärjestelmä osaa päätellä puuttuvat tyypit annettujen perusteella. Tyyppien päättely tekee ohjelmista huomattavasti tiiviimpiä ja selkeämpiä, kun tyyppinotaatiota ei ole pakko viljellä joka paikkaan. Tosin etenkin funktioita määriteltäessä on hyvä käytänne lisätä myös tyyppinotaatio, koska se selkeyttää ohjelman ymmärtämistä ja toimii dokumentaationa. Yksinkertainen esimerkki tyyppien päättelystä:

Listing 3.3. Tyyppien päättely

```
-- funktio ilman tyyppinotaatiota
onkoA x = x == 'a'

-- funktio tyyppinotaatiolla
onkoB :: Char -> Bool
onkoB x = x == 'b'
```

Esimerkissä 3.3 molemmat määritellyt funktiot “onkoA” ja “onkoB” ovat kääntäjän mielestä oikein, koska se pystyy päättelemään molemmissa tapauksissa oikeat tyypit. Jälkimmäisessä funktiossa on käytetty tyyppinotaatiota funktion määritelmässä, jossa määritellään funktion parametrin olevan yksi “Char” eli kirjain ja paluuarvon olevan “Bool” eli totuusarvo. Tyyppijärjestelmä osaa päätellä ensimmäisen funktion tyypin aloittaen varmasti tiedetyistä tyypeistä, eli tässä tapauksessa ’a’ kirjaimesta, jolle se antaisi tyypiksi “Char”. Sitten tyyppijärjestelmä käyttää tietoa vertailusta “==”, jossa tyyppien on oltava samoja. Siten myös arvon “x” täytyy olla tyyppiä “Char”. Funktion parametri on siis tyyppiä “Char” ja paluuarvo “==” palauttama totuusarvo “Bool”.

3.2.2 Tyypit ja tyyppimuuttujat

Haskellissa on yleisiä yksinkertaisia tyyppejä, kuten “Char, Int, String” kirjaimille, kokonaisluvuille ja merkkijonoille. Funktioilla on myös tyypit, kuten esimerkissä 3.3 tuli esille. Lisää esimerkkejä on listattu alla 3.4:

Listing 3.4. Esimerkkejä funktioiden tyypeistä

```
-- Yksinkertaisia funktiotyyppejä:
not :: Bool -> Bool
chr :: Int -> Char
ord :: Char -> Int

-- Tyyppimuuttujia hyädyntäviä funktiotyyppejä:
length :: [a] -> Integer
head :: [a] -> a
fst :: (a,b) -> a
map :: (a -> b) -> [a] -> [b]
```

Koodiesimerkissä 3.4 on listattu tyyppimuuttujia hyödyntäviä funktioita. Funktion “head” tyyppimääritelmä käyttää tyyppimuuttujaa “a”. Tämä tarkoittaa sitä, että a:n paikalle voidaan laittaa mikä tahansa tyyppi. Jos funktiota “head” kutsuttaisiin listalla merkkijonoja tyypiltään “String” niin silloin voi kuvitella funktion tarkemman tyypin olevan “head :: [String] -> String”. Tyyppimuuttujilla voidaan siis yleistää funktioden tyyppejä toimitaan yleisemmällä

tasolla. Tyyppimuuttujia käytetään paljon, koska usein ei ole kannattavaa rajoittaa funktiota tarkasti vain tietyillä tyypeillä toimiviksi. Useissa tilanteissa ei kuitenkaan ole mahdollista toteuttaa funktiota toimimaan yleisesti millä tahansa tyypillä “a”. silloin hyödynnetään tyyppiluokkia.

Tyyppiluokat ovat hyvin samankaltaisia kuin monien muiden kielten geneerisyys, mutta Haskellissa ne ovat paljon voimakkaampia, koska ne mahdollistavat helpon tavan kirjoittaa erittäin geneerisiä funktioita jos ne eivät käytä tiettyjä tyyppien ominaisuuksia. Tyyppimuuttujia käyttäviä funktioita kutsutaan polymorfisiksi funktioiksi. (Lipovaca 2011)

3.2.3 Tyyppiluokat

Tyyppiluokka on eräänlainen rajapinta, joka määrittelee käytöstä. Tyypin kuuluessa tyyppiluokkaan se tukee ja toteuttaa tyyppiluokan määrittelemän toiminnallisuuden. Useat olio-ohjelmoinnista tulevat käyttäjät ovat hämmentyneitä tyyppiluokista, koska he luulevat niitä samaksi konseptiksi kuin olio-ohjelmointikielissä esiintyvät luokat. Niitä ne eivät kuitenkaan ole vaan ne muistuttavat enemmänkin esimerkiksi Java-ohjelmointikielen rajapintoja, mutta parempia. (Lipovaca 2011; Hall ym. 1996)

Esimerkiksi useat erilaiset lukutyypit kuuluvat “Num”-tyyppiluokkaan ja siten monet lukuja käsittelevät funktiot käyttävät tyyppiluokkaa tyyppimääritelmässään, kuten alla nähtävissä funktioissa 3.5. Num tyyppiluokka on hyvä esimerkki siitä, kuinka tyyppiluokkien avulla päästään yleisemmälle tasolle funktioiden määrittelyssä ja yksi funktio voi niiden avulla toimia monenlaisille luvuille. Tämä johtaa yksinkertaisempaan toteutukseen, kun samalle funktiolle ei aina tarvitse olla useaa toteutusta.

Listing 3.5. Esimerkkejä tyyppiluokista

```
abs :: (Num a) => a -> a
(*) :: (Num a) => a -> a -> a
(==) :: (Eq a) => a -> a Bool
(>) :: (Ord a) => a -> a -> Bool
```

Esimerkissä 3.5 on tutun “(==)”-funktion yleisempi määritelmä. Aikaisemmin funktiota tarkasteltiin huomioimatta “Eq” tyyppiluokkaa. Tämä tyyppiluokka tarjoaa rajapinnan yhtä-

suuruuden testaamiselle. Mikä tahansa tyyppi voi käyttää tätä tyyyppiluokkaa tilanteissa joissa yhtäsuuruuden käsittely on mielekästä arvojen välillä. Tyyppiluokkien välillä voi myös olla riippuvuuksia, kuten esimerkiksi tyyppiluokka “Ord” tarvitsee osallisilleen myös tyyppiluokan “Eq” vertailuja varten.

3.2.4 Algebralliset tyypit

Haskellin tyyppijärjestelmässä on myös algebrallisia tyyppejä. Sana algebrallinen tulee mahdollisesti siitä ominaisuudesta, että algebrallisia tietotyyppejä luodaan algebrallisilla summa ja tulo -operaatioilla. Tietotyyppi “Bool” on yksinkertainen esimerkki summatyypistä ja itsemääritelty uusi tyyppi “Coord” tulotyypistä. Näistä esimerkki alla 3.6:

Listing 3.6. Tulo- ja summatyyppi

```
-- Bool tyypin määritelmä
data Bool = True | False

-- Keksityn Coord tyypin määritelmä
data Coord = C Int Int
```

“Bool” on summatyyppi, koska sillä on kaksi mahdollista arvoa “True” ja “False”, jotka on eroteltu syntaksissa “|”-merkillä. Mahdolliset arvot tyypille saadaan siis laskemalla määritelmän vaihtoehtojen summa. Esimerkissä 3.6 määritellyllä “Coord” tyypillä on konstruktori “C”, joka tarvitsee kaksi kokonaislukua. Tyyppi koostuu siis kahdesta kokonaisluvusta ja kuvastaa koordinaattia. Jos kokonaisluvut olisi rajoitettu siten, että ne ovat aina positiivisia ja suurin mahdollinen arvo olisi 10, niin kaikkien mahdollisten koordinaatti tyypin arvojen määrä saataisiin kertolaskulla “11 * 11”. Tulotyypin nimitys tulee siis tavasta laskea mahdollisten arvojen lukumäärä.

Summa- ja tulotyyppiin ja mahdollisten arvojen lukumäärään palataan tutkielman kurssin asetelmaa ja aineiston analysointia käsittelevissä luvuissa, joissa käydään läpi automaattitehtäviä.

4 Aiempi tutkimus

Luvussa esitellään metodit joilla lähdekirjallisuutta on haettu sekä aiheeseen liittyvä aiempi tutkimus. Ensiksi käydään läpi aiempaa tutkimusta ohjelmoinnin opetuksesta keskittymällä funktio-ohjelmointiin ja automaattitehtäviin. Seuraavaksi esitellään Haskelliin liittyvän opetuksen tutkimuksen tuloksia. Lopuksi käsitellään tyyppeihin ja tyyppijärjestelmiin liittyviä tutkimuksia.

Lähteiden etsimiseen käytettiin kolmea eri hakumetodia. Avainsanahaku on metodi jossa lähdekirjallisuutta etsitään tietyillä avainsanoja liittyen tutkimusaiheeseen. Taaksepäinhaku on metodi jossa uutta lähdekirjallisuutta etsitään jo tunnetun kirjallisuuden lähdeluetteloista. Eteenpäinhaku on metodi jossa uutta lähdekirjallisuutta etsitään julkaisuista jotka viittaavat jo tunnettuun kirjallisuuteen. Taaksepäin- ja eteenpäinhakuihin kuuluu molempiin askel jossa löytyneiden tekijöiden muu kirjallisuus käydään läpi mahdollisten uusien lähteiden toivossa. Tässä tutkimuksessa avainsanahakua käytettiin alussa ensimmäisten artikkeleiden etsimiseen. Näistä artikkeleista etsittiin lisää kirjallisuutta taaksepäin- ja eteenpäinhakujen avulla. Kirjallisuuden haun lähteenä toimi pääosassa Google Scholar. Myös ACM Digital Library ja IEEE Xplore olivat käytössä kirjallisuuden haussa, mutta huomattavasti pienemmässä roolissa.

4.1 Ohjelmoinnin opetus

Tässä kappaleessa keskitytään ohjelmoinnin opetusta tutkiviin papereihin ja niistä etenkin funktio-ohjelmointia tutkineisiin. Funktio-ohjelmoinnin opetusta on tutkittu useilla alustavilla ohjelmointikursseilla. Lähteissä keskitytään etenkin papereihin joissa on käytetty staattisesti tyypitettyä ohjelmointikieltä. Funktio-ohjelmoinnin opetusta ensimmäisen vuoden opiskelijoille on tutkittu seuraavissa papereissa.

Findler ym. 1997 tutkivat ohjelmoinnin opetusta Scheme funktio-ohjelmointikielellä ja tässä paperissa kuvaavat huomioitaan DrScheme-ympäristön vaikutuksista oppimiseen. DrScheme on kehitysympäristö Scheme ohjelmointikielille. Paperi keskittyy muuten lähinnä Schemeen ja DrScheme-ympäristöön, mutta johtopäätöksissä mainitaan kuitenkin, että

staattisesti tyypitetyt kielet voisivat myös hyötyä graafisista tyyppivirheiden selityksistä. Muuten paperi ei tyyppijärjestelmiin ota kantaa, koska Schemessä tyypit eivät ole olennaisena osana sen kuuluessa Lisp-kieliin.

Joosten, Van Den Berg ja Van Der Hoeven 1993 tutkivat ohjelmoinnin johdantokurssin muuttamista funktio-ohjelmointikurssiksi. Heidän motivaationaan oli selvittää funktionaalisen kielen vaikutusta kurssia käyvien oppilaiden suoriutumiseen. He toteavat, että ohjelmoinnin johdantokurssin laatu on parantunut ja oppilaat oppivat käyttämään abstraktiota suunnitteluvälineenä ja osaavat kuvata ongelmat formaalisti. He toteavat myös, että oppilaat ratkaisevat enemmän ja haastavampia ongelmia kuin aikaisemmin. Joosten, Van Den Berg ja Van Der Hoeven 1993 ensimmäisen vuoden ohjelmointikurssilla käytetty kieli oli Miranda, joka on ollut yksi vahvoista motivaatioista ja ideoista Haskellin takana. Miranda muistuttaa hyvin paljon Haskellia ja siten tutkimuksen havainnot ovat mielenkiintoisia tässä tutkielmassa vaikka kyseinen paperi on julkaistu jo vuonna 1993. Vaikka tulokset olivat positiivisia, niin he tunnistivat kolme selkeää haastetta: assosiatiivisuus, tyyppimääritelmät ja laskentamalli. Assosiatiivisuuden kanssa opiskelijoilla oli ongelmia funktioaplikaatioissa kun kielen syntaksi mahdollisti funktioiden ketjuttamisen ilman sulkuja. Kielen laskentamalliin liittyvät haasteet juontuivat imperatiivisistakielistä, kun oppilaat yrittivät ratkaista imperatiivisella mallilla kurssin funktio-ohjelmoinnin pulmia.

Tyyppimääritelmiin liittyvät haasteet ovat tämän tutkielman kannalta kaikista mielenkiintoisimpia ja Joosten, Van Den Berg ja Van Der Hoeven 1993 tunnistivat niistä kolme kategoriaa. Ensimmäinen näistä on annetun tyyppimääritelmän ymmärtäminen hankaluus, kuten määritelmä “a -> b -> c” ymmärretään seuraavasti “((a -> b) -> c)” tai tyyppimääritelmän rakennetta ei tunnisteta ja esimerkiksi “a -> b -> c” määritelmä tulkitaan 3 argumentin funktiona. Toinen tunnistetuista kategorioista on haaste tyypin määrittelyssä tietyille funktioille. Tästä kirjoittajat antavat esimerkkeinä haasteet käyttää sulkeita argumenttien ympärillä, jotka ovat funktioita, useamman argumentin funktioita ei tunnistettu ja liian rajoittavat tai liian laajat tyyppimääritelmät funktioilla. Kolmanteen kategoriaan paperin kirjoittajat keräsivät sekalaiset virheet, joista erikseen on mainittu tyyppivirheiden viestien väärinymmärrykset. Opiskelijat eivät usein käyttäneet virheilmoituksen sisältöä hyödykseen vaan lähinnä siinä ilmoitettua virheen sijaintia. Tyyppeihin liittyvät ongelmat olivat kymmenen yleisim-

män haasteen joukossa. Kurssin muokkaamisen ja palautteen arvioinnin jälkeen tyyppeihin liittyvät ongelmat saatiin poistettua kymmenen yleisimmän haasteen joukosta myöhemmillä kursseilla.

Clack ja Myers 1995 kokosivat kokemuksia, ongelmia ja ratkaisuehdotuksia funktio-ohjelmoinnin opetuksesta. He kuvaavat funktiokielten vapauttavan opiskelijat monimutkaisesta syntaksista, semantiikasta ja muistinhallinnasta mahdollistaen keskittymisen ongelmien ratkaisemiseen. Funktio-ohjelmointi ei kuitenkaan ole universaali yleislääke. Opiskelijoilla on silti ongelmia kielen ominaisuuksien, ohjelmien konseptien ja imperatiivisen paradigman -taakan kanssa. Tässä paperissa he esittelevät havaitsemiaan tyypillisiä oppilaiden ongelmia liittyen funktiokielten opetukseen. Kieleen liittyvät ongelmat on jaettu seuraaviin kategorioihin: tyypit, rekursio, akkumulointi (accumulate), listat ja korkeamman tason funktiot. Tämän tutkielman kannalta näistä kiinnostavimpia ovat tyyppeihin liittyvät ongelmat. Clack ja Myers 1995 kuvaavat neljää tapausta liittyen ongelmiin tyyppien kanssa. Totuusarvon väärinkäyttö siten, että käyttää “true” ja “false” merkkijonoja “bool” tyypin sijaan tai ylimääräisten tarkistusten käyttö. Oppilailla oli myös hankaluuksia ymmärtää numerotyyppien ero merkkijono numeroihin. Funktioiden tyyppimääritelmät aiheuttavat myös ongelmia ja monilla oppilailla on vaikeuksia hyväksyä funktion tyypin olevan kuvaus tyypistä toiseen. Opiskelijat ajattelevat tämän olevan kohdetyyppi ja se johtaa isoon osaan funktioiden tyyppimääritelmien virheistä. Clack ja Myers 1995 yrittivät saada oppilaat kirjoittamaan ensin funktiolle tyyppimääritelmän ja sitten vasta toteutuksen, mutta useat oppilaat jättävät määritelmän kirjoittamatta, koska uskovat sen vain luovan virheitä. Viimeisenä tyyppeihin liittyvistä virheistä Clack ja Myers 1995 nostavat esiin tyyppivirheet ja niiden sekavuuden aloittelevalle oppilaalle. Oppilaat päätyvät yleensä lukemaan virheviestistä vain virheen sijainnin lähdekoodissa ja yrittävät mennä sinne suoraan ratkomaan ilman tyyppivirheen tarkempaa ymmärtämistä. Tämän huomasivat myös edellisen paperin kirjoittajat Joosten, Van Den Berg ja Van Der Hoeven 1993.

Chakravarty ja Keller 2004 tutkivat funktio-ohjelmoinnin opettamisen riskejä ja hyötyjä opiskelijoiden ensimmäisenä opiskeluvuotena. He toteavat, että funktionaaliset ohjelmointikielet helpottavat ohjelmoinnin perustekniikoiden ilmaisua, ohjelmoinnin oleellisten konseptien esittämistä ja analyyttisen ajattelun kehittymistä sekä ongelmanratkaisu-

taitojen kasvattamista. Chakravarty ja Keller 2004 käyttivät kursseillaan ohjelmointikielenä Haskellia. He esittelivät tyypit ja tyyppiluokat heti alusta alkaen painottaen niitä koko kurssin ajan. Painotuksen takana on ajatus tyyppien tärkeydestä ongelmien ratkaisussa, niiden tuoma arvo sovelluskehityksessä ja etenkin niiden ollessa keskeinen konsepti tietojenkäsittelyssä. Heidän tyyppeihin kohdistunut painotuksensa näkyy paperin tuloksissa, joissa ei mainita hankaluuksista tai ongelmista. Tosin paperi ei kuvaile tarkemmin kurssin tehtäviä, joten tästä ei voi tehdä suurempia johtopäätöksiä.

Tirronen ja Isomöttönen 2012 tutkivat itseohjautuvaa oppimista funktio-ohjelmointikurssilla keskittyen oppimateriaalien suunnitteluun. Paperin kurssi on sama, kuin miltä tämän tutkielman tiedot on kerätty, mutta aiemmalta vuodelta. Paperissa he esittelevät ajatuksia ja vaikutteita kurssimateriaalin takana. Tämä antaa hieman näkökulmaa tehtävien tyyliin ja mitä niillä pyritään saavuttamaan. Materiaalien suunnittelu perustuu pitkälti “Cognitive load” nimiseen teoriaan ja tätä on avattu paperissa tarkemmin. Siinä myös käydään esimerkin avulla läpi kuinka tätä kyseistä teoriaa voi hyödyntää materiaalien suunnittelussa.

Brown ja Altadmri 2014 tutkivat aloittelijoiden ohjelmointivirheitä vertaamalla kurssien tietoa ja tuloksia opettajien uskomuksiin. Tutkimuksen kurssit keskittyivät lähinnä Java-ohjelmointikielellä toteutettuihin ohjelmointikursseihin ja siten virheitä tarkasteltiin osittain olio-ohjelmoinnin näkökulmasta. Syntaksi- ja tyyppivirheet olivat yleisimmät tunnistetut virheet. Vaikka nämä tulokset eivät Haskellin tyyppijärjestelmään liity, niin ne kuitenkin tukevat viitteitä siitä että useissa staattista tyypitystä käyttävissä kielissä on uusilla opiskelijoilla haasteita tyyppijärjestelmän kanssa ohjelmointiparadigmasta riippumatta.

4.2 Automaattitehtävät

Tässä kappaleessa tutustutaan lyhyesti automaattitehtäviä koskevaan tutkimukseen, koska iso osa tähän tutkielmaan liittyvistä kurssin tehtävistä on automaattitehtäviä. Papereista pyritään etsimään huomioita automaattitehtävien vaikutuksista kurssiin tai opiskelijaan, jotta mahdolliset vaikutukset osataan ottaa huomioon tutkielman aineistoa analysoidessa.

Automaattitehtävät tarkoittavat tehtäviä, joissa opiskelija voi tarkistaa ratkaisunsa, siten että saa siitä automaattisesti tarkastetun tuloksen. Usein automaattitehtävät on toteutettu es-

imerkiksi web-sovelluksena, jolloin opiskelija voi syöttää ratkaisunsa tarkistettavaksi selaimella tai jopa tehdä koko tehtävän interaktiivisesti. Automaatiolla voi olla eri tasoja, mutta sen tärkein idea on tarjota viiveetöntä palautetta ja keventää opettajan taakkaa. Se voi olla osittain automatisoitua, jolloin opettaja saa esimerkiksi valmiiksi formatoidun ja yksinkertaistetun tiedoston osittaisilla tarkastuksilla tarkastettavaksi. Osan tehtävistä tarkastusprosessi taas on mahdollista automatisoida kokonaan.

Saikkonen, Malmi ja Korhonen 2001 tutkivat automaattitehtävien vaikutuksia ohjelmointikurssilla. Kurssilla oli käytössä kieli nimeltä Scheme. Tutkimuksessa tuli selville oppilaiden positiivinen suhtautuminen automaattitehtäviin ja paperissa avattiin myös järjestelmän arkkitehtuuriin liittyviä huomioita.

Malmi, Korhonen ja Saikkonen 2002 ovat keränneet tietoa 10 vuoden ajan ohjelmoinnin alkeiskurssilta, tietorakenteiden kurssilta sekä algoritmit kurssilta. Tässä paperissa he esittävät kokemuksiaan liittyen automaattitehtäviin, jotka ovat täysin automaattisia, eli eivät vaadi opettajalta mitään toimia arvioinnissa. Oppilaiden virheisiin ei ole kuitenkaan tässä paperissa vielä keskitytty, vaan huomioit koskevat enemmänkin yleistä oppimista ja motivaatiota tehdä kurssitehtäviä. Malmi, Korhonen ja Saikkonen 2002 huomioivatkin automaattitehtävien kannustaneen oppilaita yrittämään ratkaisuja niin monta kertaa että olivat tyytyväisiä lopputulokseen ja siten ne kannustivat oppilaita tekemään enemmän töitä. Automaattitehtävät myös kevensivät opettajan taakkaa.

Malmi ja Korhonen 2004 analysoivat TRAKLA nimisen järjestelmän automaattitehtävien palautuksia ja vertasivat kahta erilaista palautuspolitiikkaa. Aikaisemmin kurssilla oli rajattu määrä palautuksia tehtäville, mutta uudemmassa versiossa he sallivat rajattoman määrän palautuksia sillä muutoksella, että tehtävän tiedot muuttuivat. Tarkoittaen sitä että tehtävän vaikeustaso ja tavoite pysyivät samana, mutta käytetyt arvot esimerkiksi muuttuivat aina palautuksen jälkeen. Täten tehtäviä ei voinut läpäistä väkisin arvaamalla. Malmi ja Korhonen 2004 huomasivat, että ensimmäisten palautusten taso laski huomattavasti, kun palautusten rajoitus poistettiin. He huomasivat myös oppilaiden oppineen paremmin, koska oppilaat tekivät samanlaisen tehtävän eri arvoilla useampaan kertaan yrittäessään ratkaista sitä. Näiden havaintojen perusteella on siis hyvä pitää mielessä virheiden mahdollinen lisääntyminen automaattitehtävissä, joissa ei ole palautuksilla rajoituksia.

Karavirta, Korhonen ja Malmi 2005 tutkivat uudelleenlähetystä tukevista automaattitehtävistä kerättyä tietoa. He käyttivät klusterointia erottaakseen erilaisia oppimisryhmiä ja niiden käyttäytymistä uudelleenlähetyksien ja pisteiden suhteen. Näitä ryhmiä vertaamalla keskenään he päättelivät pienellä osalla oppilaista olevan riski käyttää uudelleenpalautuksia epätehokkaasti. Täten he rajoittivat uudelleenpalautusta siten, että tehtävät alkuarvot muuttuvat aina uudelleen arvioitaessa. Karavirta, Korhonen ja Malmi 2005 havaitsivat pitämällään kurssilla olevan n. 13% oppilaista tyyppiä “iteroija”. Tämän ryhmän uudelleen palautusten määrä on huomattavasti suurempi kuin muilla paperissa esitellyillä ryhmillä. Tässä tutkimuksessa on siis hyvä pitää mielessä mahdolliset iteroijat, koska jos he ovat vain osan kurssista paikalla voivat jotkut alkupään tehtävät vaikuttaa huomattavasti vaikeammilta virheiden määrän perusteella. Jos taas vertailua tehdään tehtävä kohtaisesti, niin tältä mahdollisesta poikkeamalta voidaan välttyä.

Brusilovsky ym. 2014 kartoittivat automaattitehtäviä hyödyntäviä järjestelmiä. Paperi keskittyy yleisemmällä tasolla opetukseen ja interaktiiviseen opetusmateriaaliin, mutta on hyvää taustatietoa automaattitehtävien toteutuksista vaikka ei vahvasti liitykään tähän tutkielmaan.

4.3 Haskellin opetus

Ohjelmoinnin opetusta tutkivia papereita esiteltiin aikaisemmin, mutta tämä kappale esittelee papereita, joissa on tutkittu funktio-ohjelmoinnin opetusta Haskell-ohjelmointikielellä. Huomiot seuraavista papereista ovat siis hyvin olennaisia tämän tutkielman kannalta. Kappaleessa nostetaan papereista esiin tämän tutkielman kannalta mielenkiintoisimpia kohtia, eli erityisesti havaintoja tyyppeihin liittyvistä virheistä ja niiden mahdollisista syistä.

Tirronen, Uusi-Mäkelä ja Isomöttönen 2015 tutkivat aloittelijoiden virheitä Haskell-ohjelmointikielellä. Kurssi jolta tilastot ja tiedot kerättiin on sama kuin tässä tutkielmassa. Näitä tietoja kerätessä kurssille oli ilmoittautunut 88 oppilasta, joista 55 palautti ensimmäisen tehtävän. Suurella osalla oppilaista oli jo ohjelmoinnin peruskurssi suoritettuna. Kurssilla käsiteltiin funktio-ohjelmoinnin ja Haskellin perusasioita. Kaikki palautetut tehtävät tallennettiin ja analysointia varten niitä siistittiin ja karsittiin turhia, kuten duplikaatteja pois. Tirronen, Uusi-Mäkelä ja Isomöttönen 2015 painottavat paperissa esiin-

tyvien trendien väkisinkin johtuvat valituista tehtävistä. He mainitsevat esimerkkinä etenkin yhden tehtävän, joka testasi oppilaiden ymmärrystä tyyppijärjestelmästä vaikuttaneen huomattavasti tyyppeihin liittyvien virheiden määrään tilastoissa. Tyyppeihin liittyneistä virheistä oli 69% väärinymmärryksiä tyyppien välillä ja 33% näistä virheistä aiheutui tästä yhdestä tehtävästä.

Tirronen, Uusi-Mäkelä ja Isomöttönen 2015 jakoivat tyyppeihin liittyvät virheet seuraaviin kategorioihin, joita ei kannata kääntää: CouldntMatch, NoInstance, CandDeduce ja InfiniteType. Koodi CoudntMatch esittää virheitä joissa operaatio odotti tietynlaista tyyppiä, mutta sai sopimattoman tyypin. Koodi NoInstance viittaa virheisiin joissa funktiolla on tyyppiluokan tuomia rajoitteita ja sille annetaan argumentteja jotka eivät täytä näitä rajoitteita. Koodi CantDeduce viittaa tapauksiin joissa lauseke voi olla oikein tyypitetty, mutta kääntäjä ei pysty automaattisesti päättelemään oikeaa tyyppiä. Koodi InfiniteType kuvaa tapauksia, joissa päätelty tyyppi on ääretön, kuten esimerkiksi “let x (’x’,x) in x :: (Char,(Char,(Char...”. Yhteensä noin 40% kaikista virheistä sijoittui näihin virhekategorioihin ja ne olivat läsnä 22% kaikista oppilaiden harjoitus-sessioista. 69% CouldntMatch tyyppisistä virheistä johtui siitä, kun funktiolle annettu argumentti oli väärää tyyppiä tai heikkoa ymmärrystä tyyppijärjestelmästä. Kolmasosa näistä sessioista liittyi tehtävään, joka erityisesti testasi ymmärrystä tyyppijärjestelmästä. Muista sessioista, joissa tyypit eivät vastanneet oli yleisin virhe erehtyä sekoittaa alkio ja lista alkioita. Muut CouldnMatch koodin virheet liittyivät arvojärjestykseen ja syntaksiin. Arvojärjestyksessä haasteena oli ymmärtää sulkeiden paikka funktioaplikaatiossa ja tätä hämmensi lisää “.” ja “$” -operaattorien esittely. Syntaksi virheissä oli yleistä Haskellin joustava syntaksi, mutta tiukka tyyppitarkastus, jolloin syntaktisesti validit ratkaisut johtivat virheisiin, koska tyyppijärjestelmä hylkäsi ne. Koodin NoInstance virheet esiintyivät polymorfisia funktioita käytettiin arvoilla joille funktiota ei ole määritelty. Tämän virheen ensisijainen syy oli Haskellin numeeristen vakioiden ylikuormitus. 61% näissä tapauksissa numeerinen vakio kirjoitettiin ei-numeerisen termin sijalle, kuten funktion tai listan. Koodien InfiniteType ja CantDeduce tyyppisiä virheitä esiintyi suhteellisen vähän ja niistä ei löytynyt selkeitä vaikuttajia.

Tirronen, Uusi-Mäkelä ja Isomöttönen 2015 huomasivat tutkiessaan virheiden korjaamisaikoja tyyppivirheiden kuluttaneen eniten opiskelijoiden aikaa. Toiseksi eniten oli

syntaksiin liittyviä virheitä ja myös hankalampia ratkaistavia ajonaikaisia virheitä oli, mutta niitä oli harvemmin joten ne veivät vähemmän aikaa kokonaisuudessaan.. Tyyppivirheiden runsas määrä johtui useista seikoista, koska Haskellin joustava syntaksi johtaa useisiin erilaisiin virheisiin, jotka raportoidaan tyyppivirheinä. Haasteet Hindley-Milner tyylisen tyyppijärjestelmän ymmärtämisestä johtavat useisiin tyyppeihin liittyviin väärinymmärryksiin ja se miten tyyppivirheet raportoidaan voi johtaa niiden vaikeampaan tulkintaan.

Tirronen, Uusi-Mäkelä ja Isomöttönen 2015 huomauttavat myös, että kolme yleistä virheviestityyppiä kattavat suuren osan aloittelijoiden kohtaamista virheistä. He epäilevät, ettei ole paljoa tehtävissä syntaksivirheiden parantamiseksi tai siihen miten kääntäjä viittaa väärinkirjoitettuihin muuttujiin, mutta tyyppivirheisiin liittyen asioita voisi parantaa. Haskellin syntaksin rakenne on erittäin joustava ja suuri osa ohjelmoijien virheistä raportoidaan tyyppivirheinä väittäen jonkin tyypin olevan epäsopiva toisen tyypin kanssa. Tyyppivirheet nousevat esiin kun oppilas erehtyy esimerkiksi Int ja String tyyppien välillä tai unohtaa antaa yhden argumentin funktiolle. Paljon hyvin erilaisia oppilaiden virheitä päätyy täten raportoitavaksi yhdenmukaisesti. He ehdottavatkin, että suurimpia virheluokkia täsmennettäisiin pidemmälle ja se olisi niin aloitteleville, kuin myös kokeneille ohjelmoijille hyödyllistä.

Tirronen, Uusi-Mäkelä ja Isomöttönen 2015 tiivistävät tutkimuksen tavoitteena olleen ymmärtää aloittelevien Haskell ohjelmoijien matalantason virheet ja tutkia kuinka kääntäjä raportoi virhe viesteillä oppilaiden virheet. Heidän tutkimuksensa paljasti useita kielen suunnittelusta juontavia syitä opiskelijoiden virheille. Näihin ongelmiin kuului liian joustavan syntaksin adoptointi, joka aiheutti ongelmia funktioaplikaatiossa, järjestyksessä ja syvästi sisennetyissä lausekkeissa. Toinen kielen suunnittelun virhe liittyen aloitteleviin opiskelijoihin on standardikirjastojen käyttämien osittaisfunktioiden tekeminen liian houkuttelevaksi valinnaksi oppilaille aiheuttaen ajonaikaisia virheitä. Viimeisenä he nostavat esiin tyyppivirheiden olleen erittäin yleisiä. He uskovat, että Haskellin tyyppijärjestelmä täytyy opettaa huolella ja eksplisiittisesti aikaisin, koska useimmat aloittelijoiden virheet ilmenevät tyyppivirheinä Haskellissa.

Heeren, Leijen ja IJzendoorn 2003 tutkivat käyttäjäystävällisemmän kääntäjän käyttämistä alustavalla ohjelmointikurssilla. He käyttivät Helium nimistä kääntäjää kurssilla selkeyt-

tämään esimerkiksi Haskell kääntäjän aloittelijoille kryptisiä tyyppivirheitä. He huomasivatkin, että suuri osa oppilaiden virheistä johtui tyyppivirheistä ja Heliumin käyttö on ollut suuri menestys kurssilla. Heliumin idea on vastaava kuin aiemmin käsitelty Findler ym. 1997 kehittämä DrScheme työkalu. Se siis tuki oppilaiden oppimisprosessia uuden kielen kanssa pyrkien ratkaisemaan yleisiä haasteita. Helium keräsi tehtävistä tietoa analysointia varten, generoi varoituksia ja vinkkejä yleisimpiin ohjelmointivirheisiin liittyen ja paransi tyyppivirheviestien selkeyttä. Heeren, Leijen ja IJzendoorn 2003 tutkimuksessa esitetyssä staattisten virheiden taulukossa huomattavasti eniten (49.9%) on "undefined variable" virheitä, toiseksi eniten (13,5%) "undefined constructor" virheitä ja loput ovatkin jo alle 10% määrissä. He huomauttavat että kaikista käännetyistä ohjelmista 46% oli kääntäjän hyväksymiä. Tähän todennäköisesti vaikuttaa ohjelmien inkrementaalinen luominen. Kääntäjän hylkäämistä ohjelmista yli puolissa tapauksista raportoitiin tyyppivirhe. Tämä jälleen korostaa ymmärrettävien tyyppivirheviestien tärkeyttä.

4.4 Tyypit ja tyyppijärjestelmät

Ohjelmoinnin ja funktiokielten opetuksen lisäksi löytyy tutkimusta myös tyyppijärjestelmien ja tyyppien opetuksesta. Tässä kappaleessa keskitytään näihin jälkimmäisiin tutkimusalueisiin. Nämä tutkimusalueet ovat hyvin kiinnostavia tämän tutkielman kannalta, koska liittyvät paljon Haskellin tyyppijärjestelmän ymmärtämiseen. Valituissa papereissa keskitytään
funktio-ohjelmointiin ja staattista tarkastusta hyödyntäviin kieliin.

Tirronen 2014 tutki oppilaiden vaikeuksia ja väärinkäsityksiä liittyen moderneihin tyyppijärjestelmiin. Tutkimuksen aineisto kerättiin funktio-ohjelmoinnin johdatuskurssilta, joka on perinteisesti vastaanotettu hyvin ja jonka osallistujamäärä on kasvanut vuosittain. Kurssin suosiosta huolimatta, opettajat ovat huomanneet jatkuvia haasteita tyyppien opetuksessa ja spekuloivat näiden vaikeuksien mahdollisesti aiheuttavan usean oppilaan kurssin keskeyttämisen. Aineistoa kerättiin monivalintakysymysten vastauksista verkkokurssimateriaalista. Oppilaiden vastaukset kooditettiin tunnistettaviksi virheiksi tai väärinymmärryksiksi. Seuraavaksi tulokset jaettiin kahteen pääkategoriaan, joista ensimmäisen ongelmat juontuivat formaaleista kielistä ja toinen koostui tyyppijärjestelmän semantiikkaan liittyvistä

virheistä. Kerätyn tiedon perusteella Tirronen 2014 esittää kaksi hypoteettista väärinkäsitystä liittyen Haskellin tyyppejä kuvailevaan koneeseen (notional machine). Ensimmäisenä he löysivät todisteita, jotka osoittivat hämmennykseen liittyen kahteen yleiseen ohjelmoinnissa käytettävään erilaiseen polymorfismimalliin. Alityypitykseen perustuvaa polymorfismia käytetään esimerkiksi Javassa ja C#:ssa, kun taas parametrista polymorfismia käytetään ML-tyylisisssä kielissä. He myös havaitsivat, että useat tyyppimuuttujiin liittyvät ongelmat voitaisiin selittää väärinymmärryksellä, jossa oppilas olettaa ohjelmoijan aikeen sisältävän semanttisen tarkoituksen ohjelman suorituksessa. He huomioivat melkein jokaisen oppilaan tehneen virheitä parametriseen polymorfismiin liittyvissä kysymyksissä ja vastaavia ongelmia he havaitsivat myös kurssin ohjattujen tapaamisten yhteydessä. Polymorfismiin liittyvät ongelmat olivat heille tuttuja jo edellisiltä kursseilta. Tämä ennakkoluulo sai heidät tutkimaan hypoteesia siitä, että oppilaat sekoittavat parametrisen polymorfismin alityypitykseen. He päättelivät tulosten perusteella vaikuttavan siltä, että oppilaat ovat sekoittaneet tyyppimuuttujat olio-ohjelmointikielissä esiintyvällä käsitteellä juuri-luokasta (root class). Esimerkki tästä on virhe palauttaa Integer tyyppiä arvona, kun funktion määritelmässä on tyyppi “[a]”. Tämä virhe on selitettävissä väärinymmärryksellä luulemalla tyyppiluokkaa juuri-luokaksi jolloin Integer olisin sen alityyppiä ja siten funktio olisi oikein. Tähän väärinkäsitykseen voivat vaikuttaa ennakkokäsitykset, joita oppilaat ovat saaneet aiemmilta ohjelmointikursseilta, jotka yleensä ovat olio-ohjelmointikielillä.

Tirronen 2014 esittää myös toisen huomion virheiden aiheutumisesta, siinä opiskelijat antavat liikaa merkitystä muuttujien nimille. Tämän hypoteesin mukaan havainnot voisivat juontaa teleologisesti sekavasta ajattelusta, sillä ohjelmoijat valitsevat muuttujien nimet niiden tarkoituksen mukaan, eri muuttujien nimien täytyy siten merkitä eri aikomuksia ja eri ohjelmia. Tirronen 2014 mukaan samaa hypoteesia voi käyttää selittämään myös oppilaiden haluttomuutta hyväksyä samaa nimeä käyttäviä tyyppimuuttujia yleisesti tunnetuilla funktioilla. Kolmantena huomiona tyyppijärjestelmien notaatioon liittyen he huomioivat oppilailla olleen vaikeuksia sopivan abstraktiotason käytössä useissa ohjelmointitehtävissä. Useat riittämättömän abstraktion tapaukset liittyvät suoraan odottamattomiin kielen ominaisuuksiin, jotka ilmenevät hallitsevina vaikeuksina parametristen tyyppimuuttujien kanssa. Oppilailla oli myös nähtävästi vaikeuksia erotella alla olevia kielen abstrakteja malleja ja toimivat rajoittuneen oppimansa perusteella. Esimerkiksi monet oppilaat eivät kyenneet tunnistamaan

samankaltaisuutta koodipätkien välillä joissa muuttujien nimiä oli vaihdettu.

Ruehr 2008 tutki tyyppien ja funktioiden opettamista funktio-ohjelmoinnin kurssilla. Hän tekee johtopäätöksen, että tyyppien ja funktioiden oppiminen on yksi vaikeimpia osia funktio-ohjelmoinnin oppimisessa. Etenkin oppilaille joilla on heikko tausta matematiikassa ja formaaleissa notaatiossa.

5 Kurssin asetelma

Luvussa esitellään ensiksi kurssin asetelma ja kuvaillaan kurssia yleisesti. Sitten käsitellään tehtäviä, oppimisympäristöä ja automaattitehtäviä.

Kurssia on järjestetty jo useampana peräkkäisenä vuotena kahdessa osassa. Ensimmäisestä osasta on myös tullut vuonna 2017 pakollinen Tietotekniikan opiskelijoille. Tosin laajuutena 1-3 opintopistettä. Tämän kurssin voi suorittaa eri opintopistemäärillä, koska kurssi on jaettu toistensa päälle rakentuviin moduuleihin. Täten kurssia voi suorittaa niin paljon kuin ehtii tai osaa. Kursseista tarkemmin seuraavassa alaluvussa.

Kurssin materiaali on verkkomateriaalina omalla verkkosivullaan, jonne kirjaudutaan sisään. Verkkomateriaalissa on paljon tehtäviä, joista osa on upotettuja kysymyksiä oppimismateriaaliin ja osa erikseen palautettavia ohjelmointitehtäviä. Iso osa ohjelmointitehtävistä on mahdollista tehdä sivuston omassa ohjelmointiympäristössä. Tämä ohjelmointiympäristö antaa myös palautuksista hieman palautetta, jonka pohjalta omaa ratkaisua voi parantaa.

5.1 Yleistä kurssista

Tutkimuksen aineisto kerättiin 1-5 opintopisteen funktio-ohjelmoinnin kurssilta Jyväskylän yliopistolla. Kurssilla käytetään Haskell-ohjelmointikieltä keskittyen funktio-ohjelmoinnin konsepteihin, mutta Haskellin käyttö vaatii kielikohtaisten asioiden opettelua, etenkin tyyppijärjestelmän osalta. Kurssilla käydään läpi aiheita yhteisistä funktio-ohjelmoinnin rakenteista, kuten rekursiosta ja foldeista Hindley-Milner tyyliseen tyyppijärjestelmään. Kurssin jälkimmäinen osuus kattaa haastavampia aiheita, kuten tyyppiluokkia ja applikatiivisia funktoreita. Kurssi kestää 8 viikkoa ja on aikaisempina vuosina ollut valinnainen, mutta viimeisimpänä vuonna siitä on tehty Tietotekniikan opiskelijoille pakollinen 1-3 opintopisteen laajuudessa. Kurssille osallistuu myös muita pääaineita opiskelevia pienissä määrin, mutta isolla osalla opiskelijoista on CS1 ja CS2 -kurssit suoritettuna. Kurssi on yleisesti hyvin vastaanotettu ja osallistujamäärät ovat kasvaneet vuosi vuodelta. Osallistujamääriä ja ensimmäisen ohjelmointitehtävän palauttaneiden oppilaiden lukumääriä on listattu alla olevaan taulukkoon 1.

Vuosi	2012	2013	2014	2015	2017
Ilmoittautuneita	55	88	167	204	159
Ensimmäisen tehtävän palauttaneita	45	30	44	88	51

Taulukko 1. Kurssin osallistujamääriä

Hyvästä vastaanotosta ja suosiosta huolimatta kurssilla on ollut toistuvasti haasteita etenkin tyyppien opetuksessa ja Tirronen 2014 spekuloikin, että nämä vaikeudet tyyppien kanssa johtavat useiden oppilaiden keskeytykseen. Tämä kyseinen funktio-ohjelmoinnin kurssi nojaa vahvasti opiskelijoiden itseohjautuvaan oppimiseen. Kurssi mahdollistaa hyvin vapaamuotoisen aikataulutuksen ja itsenäisen opiskelun, mutta osa oppilaista ei luonnostaan ole itseohjautuvia ja joillain on hankaluuksia tahdittaa omaa opiskeluaan. Tämä johtaa usein siihen, että osa opiskelijoista siirtää vaikeilta tuntuvien konseptien kuten tyyppijärjestelmän opettelua myöhemmäksi. Kurssimateriaalissa on monivalintakysymyksiä teoriaosuuksien yhteydessä, joilla voi tarkistaa osaamistaan aktiivisesti. Kurssilla on myös ohjattuja tilaisuuksia, joissa voi käydä avustetusti tekemässä tehtäviä ja selvittämässä epäselvyyksiä.

5.2 Tehtävät ja Automaattitehtävät

Alaluvussa esitellään käsiteltävät tehtävät ja automaattitehtävät.

Iso osa kurssin tehtävistä on automaattitehtäviä ja oppilaat saavat näistä nopeaa palautetta. Automaattinen palaute näky opiskelijoille verkkopohjaisen koodi kehitystyökalun ja Haskellin kääntäjän kautta. Järjestelmä tuottaa raportteja dokumentoiden käännösvirheet, tehtävän tarkastukseen käytettävien testien tulokset ja yksinkertaisen ohjelman rakenteen staattisen analyysin tulokset. Oppilaiden vastaukset tallennetaan järjestelmään arviointia varten ja tässä tutkielmassa käytetään pientä osaa tallennetuista vastauksista koostuvasta tietokannasta.

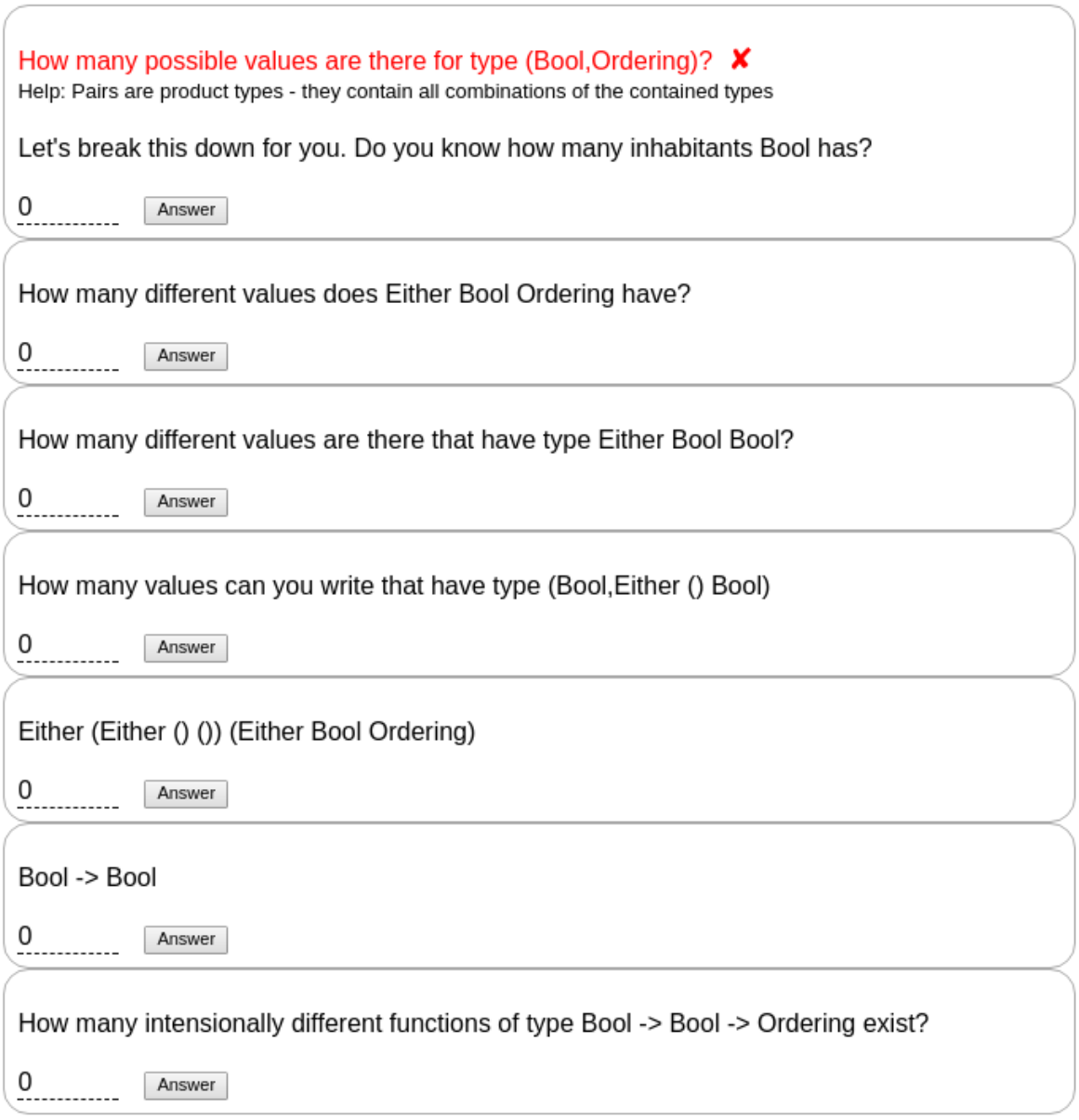
Tutkielmaan valitut tehtävät ovat olleet kurssilla käytössä kolmena vuotena, joten näiltä vuosilta on kertynyt vastauksien tietoja. Tutkielmassa keskitytään kahteen tyyppien ymmärtämistä käsittelevään tehtävään, joista ensimmäisessä on useita tyyppejä ja opiskelijan on vastattava numerolla kaikkien mahdollisten erilaisten arvojen määrä tyypille. Jos opiskelija

vastaa yhteen väärin tarjoaa tehtävä samaa konseptia testaavaa helpompaa tehtävää opiskelijalle. Tämän tarkoituksena on tukea opiskelijan oppimista. Kaikki vastaukset tallentuvat kurssin tietokantaan, josta voidaan etsiä tilastollisesti vaikeimpia kysymyksiä. Toinen tutkielmaan valittu tehtävä on ohjelmointitehtävä, jossa oppilaan pitää täydentää funktiotyyppejä vastaavat funktiototeutukset. Tämäkin tehtävä testaa vastaavien asioiden oppimista kuin edellä mainittu valintatehtävä. Ensimmäisestä tehtävästä analysoituja tuloksia pyritään varmentamaan tämän toisen tehtävän palautettuja ohjelmia tutkimalla.

Ensimmäinen tehtävä on upotettu kurssimateriaaliin, joka on verkkosivulla. Tehtävässä on lista kysymyksiä ja niihin tulee vastata numerolla. Tehtävässä on tarkoitus päätellä erilaisten mahdollisten arvojen määrä annetulle tyypille. Esimerkkinä on annettu tyyppi “Bool”, jolla on kaksi mahdollista arvoa “False” ja “True”. Tällöin parilla “Bool” tyyppejä “(Bool, Bool)” on neljä erilaista mahdollista arvoa “(True,True) (True,False) (False,True) (False,False)”. Jos vastaus on väärin, niin tehtävä tarjoaa apukysymyksiä jokaiseen erilliseen kohtaan. Apukysymyksiä on 1-4 kappaletta kohdasta riippuen. Apukysymysten tarkoitus on olla helpompia kuin pääkysymyksen ja auttaa vastaajaa ymmärtämään pääkysymyksellä haettua konseptia. Alla kuvakaappaus 1, jossa ensimmäiseen kysymykseen on vastattu väärin ja yksi apukysymys on näkyvissä.

Tehtävänanto englanniksi (kurssilla käytety kieli):

> Here is an exercise you probably won’t encounter in real life. Consider the fact that there are two possible values that have the type Bool: True and False. Obviously, then a pair of Bools, (Bool,Bool) has 4 different values: (True,True) (True,False) (False,True) (False,False).
>
> How many proper values do the following types have? The following automated exercises ask for the number of different values. If you are correct, you get a green checkmark. If you answer wrong, it will ask you a different and hopefully simpler question to help you make the right choice.
>
> To make the exercise more interesting, we have used few new types. The type () that has only one value (also called ()) and the type Ordering (which is either LT,EQ, or GT) from earlier exercise.



Kuvio 1. Ensimmäisen tehtävän kuvakaappaus

Toinen valittu tehtävä on ohjelmointitehtävä, joka tehdään kurssisivuston ohjelmointiympäristössä ja palautusyritykset tallentuvat tietokantaan. Ohjelmointitehtävän oppimistavoitteita olivat esimerkiksi: ymmärrys ja kyky käyttää tyyppimuuttujia, ymmärrys kuinka polymorfismi rajoittaa arvoja ja ymmärrys funktiotyypeistä (esim. a->a) ja kuinka niitä käytetään. Etenkin funktiotyyppien käytössä oli monilla haasteita ja tästä tarkemmin alempana tuloksia analysoivassa luvussa. Tehtävä koostuu kahdesta osasta ja ensimmäiseen osaan johdatus on

seuraavanlainen:

> Here are few simple types with their definitions blanked out. Your goal is to give good and proper definitions for each type.
>
> Any definition with correct type is fine. You don have to do anything sensible.

Tehtävänanto ensimmäiselle osalle alla:

> What to do here
>
> Here are few simple types with their definitions blanked out. your goal is to give good and proper definitions for each type. Any definition with correct type is fine. You don't have to do anything sensible. Type holes
>
> As a side goal, you should also try to practise using 'type holes'. Type holes written as variables with leading underscores (ie. _1 or _helpmeout). When program is compiled, they cause an error, which indicate the type of the expression you can write in their stead. For example, in ex1 below the compiler will tell you that _1 should be replaced with value of type Ordering (such as EQ).
>
> Try out the following:
>
> set ex3 to (_a,_b). Error will appear and tell you that _a is Bool and _b is Either () Bool set the _a to True and _b to Right _c. Error will appear and tell you that _c is Bool set _c to True and you're done.

Ohjelmointitehtävän ensimmäinen osa 5.1 koostuu tyyppimääritelmistä, joissa käytetään Bool, Ordering, Either ja () -tyyppejä. Tehtävässä siis on tarkoitus täydentää "_n" paikoille jonkinlainen funktion toteutus, joka täyttää tyyppimääritelmän. Funktion ei tarvitse tehdä mitään järkevää vaan ideana on vain täydentää funktio tyyppimääritelmän mukaiseksi. Alaviivalla ('_') alkavat muuttujat käsitellään tyyppikoloina (type hole). Tyyppikolot käsitellään ohjelmaa käännettäessä siten, että ne aiheuttavat käännösvirheen joka ilmoittaa mikä tyyppi tulisi olla tyyppikolon tilalla. Esimerkiksi "_1" tyyppikolon kohdalla kääntäjä ilmoittaisi siihen kuuluvan tyypin "Ordering" (kuten "EQ"). Tyyppikolojen on tarkoitus auttaa ohjelmoijaa epäselvissä tilanteissa.

Listing 5.1. Toisen tehtävän ensimmäinen osa

```
module Exercise where
import Prelude hiding(error)

ex1 :: (Bool,Ordering)
ex1 = (True, _1 )

ex2 :: Either Bool Ordering
ex2 = Left _2

ex3 :: (Bool,Either () Bool)
ex3 = _2

ex4 :: Either (Either () ()) (Either Bool Ordering)
ex4 = _3

ex5 :: Bool -> Bool
ex5 = _4

ex6 :: Bool -> Bool -> Ordering
ex6 = _5
```

Esimerkki vastauksesta ensimmäiseen ohjelmointiosuuteen alla 5.2. Vastauksessa on täytetty funktioiden toteutukset niin, että ne vastaavat tyyppimääritelmiä, mutta esimerkiksi “ex6” kohdassa olevalla funktiolla ei käytännössä ole todennäköisesti mitään kovin hyödyllistä käyttötarvetta.

Listing 5.2. Toisen tehtävän ensimmäisen osan yksi mahdollinen vastaus

```
module Exercise where
import Prelude hiding(error)

ex1 :: (Bool,Ordering)
ex1 = (True, EQ)
```

```
ex2 :: Either Bool Ordering
ex2 = Left False

ex3 :: (Bool,Either () Bool)
ex3 = (True, Right False)

ex4 :: Either (Either () ()) (Either Bool Ordering)
ex4 = Right (Left True)

ex5 :: Bool -> Bool
ex5 = not

ex6 :: Bool -> Bool -> Ordering
ex6 _ _ = GT
```

Tehtävän toiseen osaa johdatus on seuraavanlainen:

> What's next?
>
> Hopefully the previous exercise wasn't too bad, since the real exercise is coming up next. In the next exercise, you need to do the exact same thing as you did in the previous one. The types, however, are much more abstract.
>
> As a final hint, remember that:
>
> If you need a value of a function type (such as a->b), you can write
> x -> y. Here x will have type a and you must make sure that y will have type b. If you have a function of type a->b and you need b, just call the function! If you have a function of type a->b and you need the a, you can't get from the function.

Itse tehtävänanto tehtävää tehdessä alla:

> What to do here
>
> Just fill out the blanks with values of proper types. Make use of type holes.

Ohjelmointitehtävän toisessa osuudessa 5.3 tehtävän idea pysyy samana, eli tarkoitus on täyttää funktiot niin, että funktiotyyppi täyttyy. Tämä osuus keskittyi tyyppien osalta pareihin (tuple) ja funktiohin, kuten "(a->b)". Ohjelmointitehtävän ensimmäisen osuuden jälkeen ennen tätä toista osuutta on pieni lisäohjeistus, jossa kuvaillaan seuraavan osuuden tyyppien olevan abstraktimpia (etenkin funktiotyypit) ja annetaan yleisiä vinkkejä funktiotyyppeihin liittyen.

Listing 5.3. Toisen tehtävän toinen osa

```
module Exercise where

ex1 :: (a,b) -> a
ex1 = _1

ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 = _2

ex3 :: (a -> b) -> (c,a) -> b
ex3 = _3

ex4 :: [a] -> [a] -> [a]
ex4 = _4

ex5 :: (a, b) -> (a -> b -> c) -> (a,c)
ex5 = _5
```

Esimerkki vastauksesta toiseen ohjelmointiosuuteen alla 5.4. Tässäkin vastauksessa osa funktioiden toteutuksista on erikoisen oloisia, koska täyttävät vain funktiotyypin tarpeen eivätkä välttämättä tee mitään käytännöllistä.

Listing 5.4. Toisen tehtävän toisen osan yksi mahdollinen vastaus

```
module Exercise where

ex1 :: (a,b) -> a
ex1 (x,_) = x
```

```
ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 f g = g . f

ex3 :: (a -> b) -> (c,a) -> b
ex3 f (_,x) = f x

ex4 :: [a] -> [a] -> [a]
ex4 xs _ = xs

ex5 :: (a, b) -> (a -> b -> c) -> (a,c)
ex5 (x,y) f = (x, f x y)
```

Näihin edellä esiteltyihin tehtäviin keskitytään tässä tutkielmassa, koska ne liittyvät tyyppijärjestelmiin ja ensimmäinen niistä on tehty tätä tutkielmaa varten. Kurssilla on myös paljon muita tehtäviä, joita voisi analysoida. Muita tehtäviä onkin analysoitu aiemmin esitellyissä “Aiempi tutkimus” luvun papereissa. Kurssilta löytyy varmasti myös jatkossa tutkittavaa etenkin eri ohjelmointitehtäviin liittyen. Kurssilla käytössä oleva verkkopohjainen ohjelmointiympäristö helpottaa tutkimista paljon, koska se tallentaa opiskelijan eri vaiheita palautuksista. Toki tuloksia ei voi tulkita liian kirjaimellisesti opiskelijoiden erilaisten palautustyylien takia. Osa saattaa palauttaa tiheämmin paremman palautteen saamiseksi ja osa ratkoa tehtävän yhdellä palautuksella.

6 Tutkimusmenetelmät

Tässä luvussa esitellään tutkimuksen tavoite tutkimusmenetelmien näkökulmasta. Tutkimuskysymystä tarkennetaan ja avataan hieman, jotta siihen on mahdollista vastata selkeämmin. Kysymystä avataan myös useampaan osaan selvennykseksi. Luvussa kuvataan myös tutkimusmenetelmiä keskittyen empiiriseen tutkimukseen. Empiiristä tutkimusta kuvaillaan kvantitatiivisen, kvalitatiivisen ja tapaustutkimuksen kannalta. Tässä tutkielmassa käytetään tapaustutkimusta, joka pohjustetaan tässä luvussa ja tarkennetaan siihen liittyvät käsitteet. Lopuksi luvussa käsitellään vielä aineiston keruuta keskittyen siihen miten tehtävistä on aineistoa kursseilla tallennettu ja missä muodossa se on saatavilla.

6.1 Tutkimuskysymykset

Alaluvussa kerrataan tutkielman tutkimuskysymys ja tarkennetaan sitä, jotta tutkimuksen on mahdollista vastata siihen tarpeeksi hyvin.

Alkuperäinen tutkimuskysymys “Mitkä ovat opiskelijoiden suurimpia haasteita Haskellin tyyppijärjestelmän kanssa?” ei ole tarpeeksi tarkka, joten sitä tarkennettiin kohdistumaan kahteen valittuun kurssitehtävään. Tarkennettuja kysymyksiä ovat seuraavat: “Onko valitusta tyyppien ymmärrystä testaavasta tehtävästä havaittavissa selkeitä haasteita?” ja “Löytyvätkö samat havaitut haasteet myös toisen valitun tehtävän tuloksista?”. Näiden kysymysten avulla tutkimuksen tavoite tarkentuu ja sen on mahdollista vastata paremmin tutkimuskysymykseen. Tuloksista voi lähteä etenemään laajempaan tyyppijärjestelmän haasteiden tutkimiseen tarvittaessa. Edellä esitettyjen kysymysten mahdollisten tulosten pohjalta saa ainakin pientä suuntaa seuraavalle tutkimukselle.

6.2 Empiirinen tutkimus

Alaluvussa esitetään empiirisen tutkimuksen toteutus ja metodologia. Ensiksi kerrataan tutkimuksen tavoite ja tutkimuskysymykset, sekä käsitellään tapaustutkimusta. Tämän määrällisen tutkimuksen tapaus-aineisto on koottu edellisessä luvussa esitellyn kurssin

automaattitehtävästä. Tutkittavana ilmiönä on opiskelijoiden suurimmat haasteet Haskellin tyyppijärjestelmän kanssa funktio-ohjelmoinnin kurssilla.

Tutkimuksen tavoitteena on lisätä ymmärrystä funktio-ohjelmoinnin kurssia käyvien oppilaiden haasteista Haskellin tyyppijärjestelmän kanssa analysoimalla automaattitehtävien tuloksia. Tutkimuksessa hyödynnetään tapaustutkimusta, jossa tarkastellaan ensin monivalintatyyppisen tehtävän vastauksia, josta pyritään hahmottamaan vaikeimmat osat oppilaiden vastauksien perusteella. Tehtävän tuloksista johdetut ongelmat listataan ja niitä pyritään löytämään toisesta tyyppejä käsittelevästä ohjelmointitehtävästä.

Tutkimukseen on valittu empiirinen tutkimusote, koska tutkimuksessa ei pyritä valmiiden teorioiden testaamiseen vaan uuden teorian, mallin tai käsiterakenteen luomiseen. Empiirisistä tutkimusotteista käytössä on tapaus eli case-tutkimus. Järvinen ja Järvinen 2004 kuvaavat tätä seuraavasti, case-tutkimuksissa tarkastellaan yhtä tapausta (single-case) tai useita tapauksia (multiple case). Tiedonhankintatapoina ovat, kyselyt, haastattelut, havainnointi ja arkistomateriaalin käyttö. Kerättävä tieto voi olla näin sekä kvantitatiivista että kvalitatiivista. Luonteeltaan case-tutkimus voi olla kuvailevaa, teoriaa testaavaa tai teoriaa luovaa. Viimemainittu vastaa kysymykseen: Millaisia käsiterakenteita, malleja tai teorioita voidaan löytää tietyn casen perusteella? Joskus voi käydä niin, ettei mitään teoreettisesti uutta löydykään, mutta vasen kuvaus voi silti sisältää uutta tietämystä siitä, millainen maailma on.

Tässä tutkielmassa pyritään teoriaa luovaan ja kuvailevaan case-tutkimukseen, kun pyritään löytämään toistuvia haasteita tehtävistä ja tarjoamaan pohjaa jatkotutkimukselle. Case-tutkimus sopii menetelmäksi tähän tutkielmaan hyvin, koska sen määritelmä on vapaampi ja sitä käytetäänkin usein tutkimuksissa, joissa vaaditaan tulkitsemista kerätystä aineistosta. Case-tutkimukset voivat pohjautua mihin tahansa sekoitukseen määrällistä ja laadullista aineistoa (Yin 2003). Lisäksi case-tutkimukset eivät aina tarvitse suoria yksityiskohtaisia havaintoja aineistonaan (Yin 2003). Yin 2003 mainitsee myös että case-tutkimukset voidaan toteuttaa ja kirjoittaa useilla eri motiiveilla mukaan lukien yksinkertaiset esitykset yksittäisistä tapauksista tai haluista päätyä laajempiin yleistyksiin pohjautuen tapaustutkimuksen aineistoon.

Yin 2003 mainitsee yleisenä ohjeena alustavan määritelmän tutkimuksen yksikölle ja siten tapaukselle kannattaa olla määritelty samalla tavalla kuin alkuperäinen tutkimuskysymys. Tässä tutkielmassa tarkastellaan viimeisimmän vuoden (2017) ajalta aineistoa, joten tästä yhden vuoden aineistosta tulee sopivasti yksi tapaus. Tämä tapaus sisältää kahden valitun tehtävän tulokset aineistona ja myöhemmin jos jatkotutkimusta tehdään, niin muista vuosista voi tehdä samalla tavalla omat tapauksensa ja verrata niiden tuloksia keskenään. Valittuja tehtäviä analysoidaan kvantitatiivisilla ja kvalitatiivisilla menetelmillä, joita avataan tarkemmin seuraavassa kappaleessa.

Alasuutari 1999 toteaa että kvalitatiivista ja kvantitatiivista analyysiä voidaan pitää tietyssä mielessä jatkumona, ei vastakohtina tai toisensa pois sulkevina analyysimalleina. Täten on siis tarpeellista määritellä nämä käsitteet ja tarkentaa kuinka tässä tutkielmassa niitä sovelletaan aineiston analysoinnissa. Alasuutari 1999 mainitsee myös ettei näitä kahta tutkimuksen tekemistä tulkitsevaa mallia voi kuitenkaan samastaa kahdeksi tutkimustyypiksi. Sen sijaan voidaan erottaa toisistaan puhtaasti lomaketutkimuksen oppikirjaparadigmaa noudattava tutkimus sellaisista tutkimuksista, joissa ei noudateta mitään tai vain joitakin lomaketutkimukseen liittyvistä empiiriseen tilastolliseen tutkimukseen liittyvistä normeista. Tälle ydinparadigmaa ympäröivälle alueelle sijoittuvia monenlaisia tutkimusotteita ja menetelmällisiä ratkaisuja ei voi pitää lomaketutkimuksen vastakohtina. Ne ovat kirjava joukko menetelmällisiä ratkaisuja, joissa on sovellettu niitä tai näitä lomaketutkimuksesta tuttuja metodisia ratkaisuja ilman, että koko lomaketutkimuksen metodista sääntöpakettia olisi orjallisesti noudatettu. Tässä tutkielmassa analysoidaan aineistoa tilastollisin menetelmin, kun etsitään vastauksista yleisesti vaikeimpia tehtäviä. Tätä sovelletaan molemmissa tehtävissä, mutta etenkin ohjelmointitehtävän tulosten analysoinnissa käytetään myös laadullisia menetelmiä tulosten ymmärtämiseen. Kvantitatiivisilla menetelmillä saadaan nostettua tiettyjä tapauksia esiin, mutta niiden ymmärtäminen vaatii myös kvalitatiivisia menetelmiä. Alasuutari 1999 mainitseekin että laadullisessa tutkimuksessa on tavallista soveltaa vaihtelevassa määrin muuttuja-ajattelua ja tilastollista todistelua.

6.3 Aineiston keruu

Alaluvussa esitellään aineiston keruu -menetelmät kurssilla.

Ohjelmointikurssin materiaali ja tehtävät ovat verkossa. Kurssimateriaali sisältää monivalintatehtäviä, joiden vastaukset tallennetaan. Näiden luennon teoriaa testaavien monivalintatehtävien lisäksi on myös harjoitustehtäviä, jotka ovat pääosin ohjelmointitehtäviä. Ohjelmointia vaativista tehtävistä osan voi tehdä suoraan verkossa käyttäen kurssille suunniteltua kehitystyökalua selaimella. Näistä tehtävistä kertyy enemmän tietoa, jos opiskelija palauttaa tehtävän useammin testatakseen ratkaisuaan. Osa ohjelmointi tehtävistä taas vaatii manuaalista tarkastelua, mutta ratkaisut tallentuvat järjestelmään.

Valittu tyyppijärjestelmän ominaisuuksia testaava monivalintatehtävä testaa oppilaiden kykyä päätellä erilaisten mahdollisten arvojen määrä annetuille tyypeille. Tehtävässä on useampi tyyppi ja jokaiselle voi antaa vastauksen numerona ja yksittäiset vastausyritykset tallentuvat. Tutkielman aineiston tärkeä osa koostuu tämän tehtävän vastauksista, joista pyritään etsimään tilastollisesti selkeitä haastavimpia kohtia. Monivalintatehtävästä tehtyjä havaintoja verrataan vastaavaan ohjelmointitehtävään, jonka aineisto on palautettuja ohjelmointitehtäviä.

Aineistoa kurssilla on siis kerätty kaikista tehtävistä, mutta tässä tutkimuksessa keskitytään kahden edellä mainitun tehtävän aineistoon. Aineistoa on kerätty kurssilta jo useamman vuoden ajan, mutta tässä tutkielmassa on käytössä vain viimeisimmän vuoden (2017) tuloksia. Jos tuloksissa on mielenkiintoisia piirteitä, niin silloin voi olla kannattavaa tutkia palautuksia useamman vuoden ajalta. Viimeisimmän vuoden osalta dataa on kertynyt eniten osallistujamäärän ollessa suurin tähän mennessä. Kurssin ollessa edes pienessä laajuudessa pakollinen tietotekniikan opiskelijoille tuo se mahdollisesti myös laajemman otannan erilaisia opiskelijoita kurssille.

Aineistoa voi tarkastella lokitiedostojen muodossa. Ensimmäisestä tehtävästä on kerätty lokia siten, että jokainen vastaus tallentuu aikaleiman kanssa. Lokeista voidaan jälkeenpäin koostaa tilastoja esimerkiksi vastaajien ja vastausten lukumääristä. Tilastoja analysoidusta aineistosta löytyy seuraavasta luvusta. Myös toisen tehtävän vastaukset tallentuivat lokimuodossa siten, että jokainen ohjelmointitehtävän palautus tallentuu omanaan aikaleiman kanssa. Näitä on vaikeampi automaattisesti analysoida, mutta käsin tarkasteltuna niistä voi pyrkiä tekemään johtopäätöksiä. Seuraavassa luvussa tämänkin tehtävän vastauksia analysoidaan ja pyritään löytämään vastaavia toistuvia ongelmia kuin ensimmäisessä tehtävässä.

7 Aineiston analysointi

Luvussa käsitellään kurssidatan analysointia. Ensin tarkastellaan ja analysoidaan automaattitehtävien lokitiedostoja. Seuraavaksi tarkastellaan muiden palautusten tuloksia suhteessa automaattitehtäviin. Lopuksi listataan tutkimukseen liittyviä rajoitteita.

7.1 Automaattitehtävien lokitiedostot

Alaluvussa analysoidaan automaattitehtävien lokitiedostoja.

Ensimmäinen tehtävä “Number of possible values” testaa oppilaiden ymmärrystä tyypeistä. Tehtävässä piti päätellä mahdollisten erilaisten arvojen lukumäärä annetuille tyypeille. Tehtävän kuvaus esiteltiin aikaisemmassa luvussa. Alla tehtävän eri kohdat, joihin viitataan kirjaimilla tulosten analysoinnissa:

> A: How many possible values are there for type (Bool,Ordering)?
>
> B: How many different values does Either Bool Ordering have?
>
> C: How many different values are there that have type Either Bool Bool?
>
> D: How many values can you write that have type (Bool,Either () Bool)
>
> E: Either (Either () ()) (Either Bool Ordering)
>
> F: Bool -> Bool
>
> G: How many intensionally different functions of type Bool -> Bool -> Ordering exist?

Tehtävästä kertyi lokitietoa seuraavasti. Oppilaiden kaikki vastausyritykset tallentuivat tästä tehtävästä. Palautetuista tehtävistä näkee oppilaan yritysten määrän per tehtävä, joka mahdollistaa yksinkertaisten tilastojen kokoamisen aineistosta. Palautuksista näkee myös milloin tehtävä on mennyt oikein. Aineistosta on siis mahdollista nähdä esimerkiksi Vastaajien kokonaismäärän, eli niiden oppilaiden osuuden jotka vastasivat tehtävään. Tästä näkee myös

tarkemmin sen, kuinka moni jätti tehtävän tekemisen kesken tietyn kysymyksen kohdalla tai luovutti tietyn kysymyksen kanssa. Aineistosta näkyvä kaikkien vastausten määrä per kysymys kertoo hyvin kysymysten haastavuudesta. Aineistosta nähdään myös kuinka monta vastausyritystä keskimäärin vaadittiin jokaiseen tehtävään. Alla olevassa taulukossa 2 näkyy näitä tilastoja oppilaiden tehtävien palautuksista:

Tehtävä	A	B	C	D	E	F	G
Vastaajia	166	165	165	160	157	162	159
Vastauksia	377	477	546	524	534	843	2133
Keskeyttäjiä	5	3	11	2	4	7	14
Apukysymyksiä	2	2	2	4	3	2	1
Oikean vastauksen keskiarvo	2.27	2.89	3.31	3.27	3.40	5.20	13.40

Taulukko 2. Vuoden 2017 typecount-tehtävän tilastoja

Yllä olevasta tilastosta näkyy tehtävien välillä olevan hieman eroja. Vaikeimpia tehtäviä olivat C, F ja G. Näistä tehtävä G oli huomattavasti haastavin väärien vastausten perusteella. Kerättyä tietoa tarkemmin tarkasteltaessa voi huomata siellä esiintyvän muutamia huomattavan isoja palautusmääriä, jotka osittain nostavat G-tehtävän keskiarvoa. Vastaajien määrä pysyi kaikkien tehtävän kohtien kesken melko tasaisena n. 165 opiskelijan määränä. Vastaajien määrä laski hieman kysymysten edetessä. Ensimmäisen (A) ja viimeisen (G) kysymyksen vastaajien määrän ero oli seitsemän opiskelijaa. Vastausten määrä oli useimmissa tehtävissä 500 vastauksen paikkeilla. Tästä määrästä erosivat helpoin tehtävä A 377 vastauksella, sekä kaksi viimeistä tehtävää F ja G, joissa määrät olivat jo selkeästi suuremmat. F-tehtävässä vastausten määrä oli 843 sen ollessa siis selkeästi aiempia tehtäviä haastavampi. G-tehtävään vastauksia kertyi huomattavasti enemmän sen ollessa kaikista tehtävistä vaikein 2133 vastauksen lukumäärällä. Keskeyttäjiä eniten oli tehtävissä C (11 kpl), F (7 kpl) ja G (14 kpl). Keskeyttäjien määränkin mukaan G-tehtävä oli haastavin opiskelijoille. Tosin G-tehtävä oli ainut tehtävä, jolla oli vain yksi apukysymys tukemassa kohdan ymmärtämistä. Voi siis olla että apukysymysten puute tai epäselvä tehtävänanto kysymyksellä on vaikuttanut tulokseen, mutta näitä pohditaan tarkemmin myöhemmässä rajoitteet-alaluvussa.

Muissa tehtävissä kysyttiin kuinka monta mahdollista tai erilaista arvoa voi annetulla tyypillä olla, mutta G-tehtävässä kysyttiin kuinka monta tarkoituksellisesti erilaista funk-

tiota on annetulla tyypillä. Tehtävä oli siis hieman eritavalla muotoiltu ja siinä kysyttiinkin hieman eri asiaa. Tehtävässä kysyttiin mahdollisten funktioiden määrää mahdollisten arvojen sijaan, joka on voinut olla monelle haastavampi ymmärtää. F tehtävässä on funktiotyyppi ja muissa aiemmissa tehtävissä on Either, (), Tuple ja Bool -tyyppejä ainoastaan. F-tehtävä oli toiseksi haastavin ja siinä esiintyy myös funktiotyyppi, kuten G-tehtävässäkin. Funktiotyypit vaikuttavat siis olevan haastavampia ymmärtää kuin muut kurssilla käsitellyt tyypit.

Seuraavissa taulukoissa 3, 4, 5, 6, 7, 8 ja 9 on listattu tehtävien pääkysymykset, apukysymykset ja näille viisi yleisintä väärää vastausta. Väärien vastausten frekvenssit auttavat hahmottamaan oppilaiden vääriä käsityksiä eri konsepteista.

Pääkysymys
How many possible values are there for type (Bool,Ordering)?
5: 20 kpl, 2: 10 kpl, 3: 7 kpl, 4: 6 kpl, 1: 4 kpl (oikea: 6)
Apukysymykset
Let's break this down for you. Do you know how many inhabitants Bool has?
6: 36 kpl, 5: 5 kpl, 7: 5 kpl, 3: 4 kpl, 4: 4 kpl (oikea: 2)
Let's break this down for you. How many values does Ordering have?
2: 10 kpl, 1: 5 kpl, 4: 2 kpl, 6: 2 kpl, 0: 1 kpl (oikea: 3)

Taulukko 3. A-tehtävän kysymysten yleisimmät väärät vastaukset

A-tehtävässä 3 yhtenä vääränä käsityksenä on pääkysymyksen väärä vastaus 5, jolle taas on selkeämpi selitys. Vastauksessa on selkeästi laskettu mahdollisten tyyppien lukumääräksi "2+3" ja ei ole ymmärretty täysin sitä kuinka "Tuple"-tyypin tapauksessa oikea vastaus olisikin "2*3". Ensimmäisen apukysymyksen vääristä vastauksista yleisin oli 6, joka on aika erikoinen vastaus Bool-tyyppiin liittyen. Tässä on mahdollisesti sekoitettu kysymys ja vastattu vielä pääkysymykseen, jonka oikea vastaus olisi 6. Muuten virheellisten vastausten frekvenssit ovat melko pieniä.

Pääkysymys
How many different values does Either Bool Ordering have?
2: 36 kpl, 3: 25 kpl, 6: 22 kpl, 1: 20 kpl, 12: 18 kpl (oikea: 5)
Apukysymykset
That's not it. Let's break this down. First, how many values did Bool have?
5: 13 kpl, 1: 8 kpl, 3: 2 kpl, 6: 2 kpl, 4: 1 kpl (oikea: 2)
Let's break this down. How many values are there in Ordering?
4: 3 kpl, 6: 2 kpl, 1: 1 kpl, 2: 1 kpl (oikea: 3)

Taulukko 4. B-tehtävän kysymysten yleisimmät väärät vastaukset

B-tehtävän 4 väärien vastausten frekvensseistä suurin on pääkysymyksen yleisin väärä vastaus 2. Tämä vastaus selittyneen sillä, että oppilaat ovat ajatelleet "Either"-tyypin palauttavan vain 2 eri arvoa "Left" tai "Right". Ei ole siis ymmärretty tyyppimuuttujia liittyen "Either"-tyyppiin. "Either a b" arvo on siis joko "Left a" tai "Right b". Toki on mahdollista että tässä on vastattu mahdollisilla "Bool"-tyypin arvoilla, joita on kaksi. Saman kysymyksen toiseksi yleisin väärä vastaus 3 voisi myös johtua vastaavasta ajattelusta, että "Ordering"-tyypillä on kolme mahdollista arvoa. Kolmanneksi yleisin vastaus 6 on jo selkeämpi, sillä siinä on laskettu mahdollisten arvojen tulo samalla tapaa kuin "Tuple"-tyypin tilanteessa tulisi tehdä. Neljänneksi yleisin väärä vastaus 1 selittyneen sillä, että oppilas on ajatellut "Either"-tyypillä olevan vain yksi mahdollinen arvo. Apukysymysten väärien vastausten frekvenssit olivat pieniä.

Pääkysymys
How many different values are there that have type Either Bool Bool?
2: 79 kpl, 1: 8 kpl, 3: 5 kpl, 5: 5 kpl, 7: 4 kpl (oikea: 4)
Apukysymykset
Nope. Do you remember how many values there are in Bool?
4: 38 kpl, 3: 11 kpl, 5: 6 kpl, 6: 6 kpl, 1: 5 kpl (oikea: 2)
And,how about Either Bool Ordering?
4: 13 kpl, 2: 9 kpl, 6: 5 kpl, 3: 3 kpl, 1: 2 kpl (oikea: 5)

Taulukko 5. C-tehtävän kysymysten yleisimmät väärät vastaukset

C-tehtävässä 5 on huomattavasti vääriä vastauksia pääkysymyksessä. Oppilaista 79 on vastannut kysymykseen arvolla 2. Tämä väärä vastaus voisi tulla samanlaisesta väärinkäsityksestä kuin edellisen tehtävänkin virhe. Eli ajatellaan “Eitherin”-tyypin palauttavan joko “Left” tai “Right”. Tässä versiossa saatetaan jopa sekoittaa vielä siten, että Either palauttaa molemmissa tapauksissa “Bool” ja “Bool”-tyypillä on vain kaksi mahdollista arvoa. Toiseksi eniten vääriä vastauksia tehtävässä on ensimmäisessä apukysymyksessä, jossa niitä on 38 kappaletta arvolle 4. Tässä on ehkä ajatuksissaan vastattu vielä pääkysymykseen tai sitten on joku suurempi väärinkäsitys siitä, että “Bool”-tyypillä olisi neljä mahdollista arvoa.

Pääkysymys
How many values can you write that have type (Bool,Either () Bool) 5: 55 kpl, 4: 25 kpl, 3: 10 kpl, 8: 7 kpl, 2: 6 kpl (oikea: 6)
Apukysymykset
No. Let’s break this down. First, how many are there for Either () Bool? 2: 24 kpl, 4: 12 kpl, 1: 11 kpl, 5: 6 kpl, 6: 5 kpl (oikea: 3)
Let’s break this down. How many values for ()? 0: 5 kpl, 2: 4 kpl, 3: 2 kpl, 4: 2 kpl (oikea: 1)
Do you remember how many values Ordering has? Ei vääriä vastauksia (oikea: 3)
Let’s break this down further. Bool had how many values again? 6: 6 kpl, 5: 4 kpl, 7: 2 kpl, 3: 1 kpl, 4: 1 kpl (oikea: 2)

Taulukko 6. D-tehtävän kysymysten yleisimmät väärät vastaukset

D-tehtävässä 6 yleisin virhe näyttää tapahtuneen pääkysymyksen kohdalla. Siihen yleisin väärä vastaus on ollut 5. Tämä väärinymmärrys johtuu todennäköisesti siitä, että on summattu “Tuple”-tyypin arvot vaikka ne tulisi kertoa oikean vastauksen saamiseksi. Toiseksi yleisin väärä vastaus pääkysymykseen oli 4. Se voisi johtua esimerkiksi aiemminkin havaittuun väärinymmärryksestä “Either”-tyyppiin liittyen, jossa sillä ajatellaan olevan kaksi mahdollista arvoa. Lopuista vääristä vastauksista esiin nousee ensimmäisen apukysymyksen yleisin väärä vastaus 2. Tämä voi tukea aiempaa havaintoa väärinkäsityksestä “Either”-tyyppiin liittyen tai voi kertoa väärinkäsityksestä liittyen “()”-tyyppiin, joka on kurssilla melko

harvinainen.

Pääkysymys
Either (Either () ()) (Either Bool Ordering)
10: 33 kpl, 6: 20 kpl, 8: 14 kpl, 5: 9 kpl, 12: 8 kpl (oikea: 7)
Apukysymykset
Let's break this down for you. How many values are there in Either () ()?
1: 14 kpl, 7: 4 kpl, 4: 2 kpl, 10: 2 kpl, 11: 2 kpl (oikea: 2)
And how many valued did () have?
7: 11 kpl, 2: 5 kpl, 6: 4 kpl, 8: 3 kpl, 5: 2 kpl (oikea: 1)
Finally, how many values did Either Bool Ordering have?
6: 12 kpl, 7: 5 kpl, 4: 3 kpl, 2: 2 kpl, 8: 2 kpl (oikea: 5)

Taulukko 7. E-tehtävän kysymysten yleisimmät väärät vastaukset

E-tehtävässä 7 väärien vastausten frekvenssit olivat pääosin matalia. Yleisimmät väärät vastaukset olivat pääkysymyksessä arvoilla 10 ja 6. Nämä vastaukset voisivat selittyä seuraavasti. Jos laskee mahdollisten tyyppien määrän osassa "(Either Bool Ordering)" (5 mahdollista arvoa) ja kertoo sen kahdella tulisi vastaukseksi 10. Eli yksi mahdollisuus on, että jostain syystä on koettu tarpeelliseksi kertoa kahdella. Vastaus 6 taas voisi selittyä sillä, että on laskettua aiemmin mainittu 5 mahdollista arvoa osalle tyyppiä ja lisätty siihen "()"-tyypin arvojen määrä eli yksi.

Pääkysymys
Bool -> Bool 2: 125 kpl, 1: 60 kpl, 3: 53 kpl, 0: 10 kpl, 5: 1 kpl (oikea: 4)
Apukysymykset
Let's try something simpler first. How many different functions of type Bool -> () are there? 2, 89 kpl, 0: 13 kpl, 3: 11 kpl, 4: 9 kpl (oikea: 1)
Let's try something simpler first. How many intensionally different functions of type () -> Bool can you write? 1: 38 kpl, 0: 3 kpl, 3: 3 kpl, 4: 2 kpl (oikea: 2)

Taulukko 8. F-tehtävän kysymysten yleisimmät väärät vastaukset

F-tehtävässä 8 nousi kaksi selkeästi yleistä väärää vastausta esille. Pääkysymykseen oli vastattu 125 kertaa arvolla 2. Tämä yleinen väärä vastaus voisi johtua siitä, että oppilaat eivät ole ymmärtäneet funktiotyyppiä tai ovat päätelleen vain paluutyypin arvon ratkaisevan. Funktiolla "Bool -> Bool" on kaksi mahdollista eri paluutyypin arvoa, mutta sillä on myös kaksi eri arvoa mahdollisina parametreina. Pääkysymyksen kaksi seuraavaksi yleisintä vastausta 1 ja 3 ovat hieman erikoisempia ja saattavat olla arvauksia sen jälkeen kun on ensin koitettu vastata 2. Ensimmäisessä apukysymyksessä oli 89 kappaletta vääriä vastauksia arvolle 2. Tässä on siis laskettu mahdollisten erilaisten parametrien määrä, mutta paluutyypin arvoja on vain yksi mahdollinen "()"-tyypillä. Toisessa apukysymyksessä yleisin väärä vastaus koskee samaa tilannetta, että on laskettu mahdollisten erilaisten parametrien määrä, muttei ole huomioitu paluutyypin arvojen määrää. Kaikissa kysymyksissä on myös vastattu arvolla 0, joka on melko mielenkiintoinen vastaus.

Pääkysymys
How many intensionally different functions of type Bool -> Bool -> Ordering exist?
12: 102 kpl, 18: 60 kpl, 11: 42 kpl, 27: 36 kpl, 13: 32 kpl (oikea: 81)
Apukysymys
Let's break this down. How about in Bool -> Ordering?
6: 216 kpl, 8: 112 kpl, 5: 110 kpl, 3: 91 kpl, 2: 72 kpl (oikea: 9)

Taulukko 9. G-tehtävän kysymysten yleisimmät väärät vastaukset

G-tehtävässä 9 vääriä vastauksia oli paljon. Etenkin apukysymyksen väärien vastausten frekvenssien perusteella nähdään, että oikeaa vastausta on etsitty samalla tavalla. Yleisimpänä väärinkäsityksenä tässä tehtävässä on ollut se kuinka mahdollisten erilaisten funktioiden lukumäärä lasketaan. Suosituin väärä tapa oli parametrien ja paluuarvojen lukumäärien tulo, jolla on saatu vastaukseksi pääkysymykseen 12 ja apukysymykseen 6. Tämä selittää todennäköisesti myös toiseksi yleisimmän väärän vastauksen pääkysymykseen. Eli kun on keksitty että apukysymyksen vastaus on 9 on koitettu laskea sen tulo mahdollisten "Bool"-tyypin arvojen (2) kanssa. Oikea vastaus laskettaisiin seuraavalla tavalla "(#Ordering^#Bool)^#Bool", jossa "#tyyppi" tarkoittaa mahdollisia eri arvoja tyypille. Näin saadaan siis oikeaksi vastaukseksi pääkysymykselle 81 ja apukysymykselle 9. Mahdollinen väärinkäsitys näkyy myös apukysymyksen neljänneksi yleisimmässä väärässä vastauksessa 3 (91 kpl). Siinä on mahdollisesti päätelty mahdollisten arvojen lukumäärän olevan funktion paluutyypin arvojen lukumäärä. Tämä sama väärinkäsitys esiintyi edellisessäkin tehtävässä, joka oli ensimmäinen funktiotyyppejä käsittelevä tehtävä tässä kokonaisuudessa.

Oppilaiden yleisimmät väärinkäsitykset liittyivät siis tulo- ja summatyyppehin, sekä funktiotyyppeihin. Etenkin "Either"-tyypin ymmärtäminen vaikutti olevan haastavaa. Toisena huomattavana haasteena olivat funktiotyypit ja niiden mahdollisten arvojen ymmärtäminen. Seuraavassa alaluvussa tarkastellaan toisen tyyppien ymmärrystä testaavan tehtävän tuloksia. Tämän ensimmäisen tehtävän pohjalta voi odottaa, että oppilailla olisi funktiotyyppien kanssa enemmän haasteita verrattuna muihin typpeihin.

7.2 Muut palautukset

Toinen tarkasteltava tehtävä “type driven exercise” testasi opiskelijoiden ymmärrystä tyyppimuuttujien käytöstä, polymorfismin rajoitteista ja funktiotyypeistä. Tehtävä jakautui kahteen osioon, joista kertyi palautuksia jokaiselle opiskelijalle. Ensimmäisessä osiossa testattiin ymmärrystä lähinnä käyttäen Bool, Ordering ja Either tyyppejä. Toinen osio keskittyi enemmän funktiotyyppien ymmärtämiseen. Tarkempi tehtävän kuvaus ja esimerkit mahdollisista oikeista vastauksista löytyvät aiemmasta kurssin asetelma luvusta.

Tehtävästä kertyi lokitietoa seuraavasti. Oppilaiden kaikki vastausyritykset tallentuivat tästä tehtävästä. Vastausyritys tapahtuu aina kun oppilas suorittaa kurssin verkkosivulla olevassa ohjelmointiympäristössä luomansa vastauksen. Palautetuista tehtävistä näkee yksittäisen oppilaan yritysten määrän, joka mahdollistaa yksinkertaisten tilastojen kokoamisen aineistosta. Koska tehtävä on ohjelmointitehtävä, niin automaattinen analysointi on hyvin haastavaa tilastojen kannalta ja tässä tehtävässä onkin kannattavampaa tutkia vastauksia lukemalla ja tekemällä niistä huomioita toistuvista virheistä.

Kerätystä aineistosta näkee vastaajien lukumäärän, eli oppilaiden määrän, jotka edes kerran yrittivät palauttaa tehtävän. Kaikkien vastausten määrä selviää myös aineistosta, mutta se ei yksinään kerro tehtävän haastavuudesta. Oppilailla on erilaisia palautustyylejä. Osa palauttaa tehtävän useaan otteeseen jokaisen pienen muutoksen jälkeen ja testaa näin ohjelmointiympäristössä ratkaisuaan. Osa taas palauttaa vain kerran tai kaksi, joten tämän vuoksi palautusten määrä ei aina kuvaa haastavuutta. Keskeyttäjien määrä selviää myös aineistosta. Nämä ovat siis oppilaita joilta löytyy aineistosta vastauksia, mutta jotka eivät ole saaneet palautettua hyväksyttyä oikeaa vastausta. Taulukossa on myös laskettu oikean vastauksen keskiarvo, joka kuvaa kuinka monta kertaa oppilas on keskimäärin palauttanut tehtävän ennen oikeaa vastausta. Tämäkään ei kerro välttämättä haastavuudesta, mutta kuvaa hieman oppilaiden palautusjärjestelmän käyttöä. Alla mainittu taulukko 10:

Vastaajia	156
Vastauksia yhteensä	1823
Oikeita vastauksia	139
Keskeyttäjiä	17
Oikean vastauksen keskiarvo	11.69

Taulukko 10. Vuoden 2017 type driven -tehtävän tilastoja

Tästä taulukosta 10 ei näe yksittäisiä haastavimpia tehtäviä, kuten edellisessä tehtävässä. Tämän toisen tehtävän analysointi vaati enemmän yksittäisten vastausten lukemista ja virheiden ja vaikeuksien ymmärtämistä sitä kautta. Kaikki palautukset (1823 kpl) käytiin läpi seuraavien havaintojen saamiseksi. Yleisiä virheitä olivat seuraavat.

Lambda-funktioiden syntaksi vaikutti olevan osalla oppilaista hieman hakusessa ja vastauksissa esiintyi turhia lambdoja osan funktioista kohdalla. Tässä voi myös toki olla mahdollisena selityksenä se, että osa oppilaista on halunnut harjoitella ja kokeilla enemmän niiden käyttöä. Alla 7.1 esimerkkejä palautuksista, joissa lambdojen käytössä oli haasteita.

Listing 7.1. Haasteita lambda-lausekkeissa

```
-- tehtävän määrittelemä tyyppi:
ex2 :: (a -> b) -> (b -> c) -> (a -> c)

-- erilaisia vastauksia:
ex2 = \a -> (\q -> w) -> \b -> (\w -> e) -> \c -> (\q->e)

ex2 = \x-> (\y -> (a->c))

ex2 = \x -> (\y->y) \z -> z

ex2 = \a -> (\q -> w) -> \b -> (\w -> e) -> \c -> (\q->e)

ex2 = (\x -> y) -> (\y -> \g ) -> (\x -> \g )

ex2 = \f -> (\g -> (\f -> \g x))
```

```
ex3 :: (a -> b) -> (c,a) -> b
ex3 = \q -> \(x,y) -> \q -> a
```

Where ja Case -lausekkeita oli vastaavasti myös turhia käytetty ja niiden ylimääräinen käyttö teki vastauksista vaikeammin luettavia. Tässäkin voi olla samaa syytä taustalla, että on vain testailtu tiettyjä rakenteita. Syynä vois myös olla luulo siitä, että funktioiden täytyisi aina laskea jotakin. Ei siis välttämättä ymmärretä funktion voivan vain palauttaa arvon sen kummemmin mitään laskematta. Alla esimerkkejä 7.2, 7.3 palautuksista, joissa on mahdollisesti koettu, että funktion täytyy laskea jotain ja siksi käytetty osittain turhia “where”-lausekkeita.

Listing 7.2. Samassa palautuksessa paljon where:n käyttöä

```
ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 fab fbc = tulos
    where
    tulos a = fbc (fab a)

ex3 :: (a -> b) -> (c,a) -> b
ex3 fab p_ca = tulos_b p_ca
    where
    tulos_b p_ca = fab ((\(x, y) -> y) p_ca )

ex4 :: [a] -> [a] -> [a]
ex4 eka_l toka_l = tulos_l eka_l
    where
    tulos_l eka_l = (\x -> x) eka_l

ex5 :: (a, b) -> (a -> b -> c) -> (a,c)
ex5 pari_ab fabc = ((\(x,y) -> x) pari_ab, tulos_pari)
    where
    tulos_pari = (\(x, y) -> fabc x y) pari_ab
```

Edellinen esimerkki 7.2 on yhdestä palautuksesta, jossa korostuu hyvin tarpeettomien

“where”-lausekkeiden tuoma kompleksisuus. Seuraava esimerkki 7.3 on koottu kahdesta erillisestä palautuksesta, joissa molemmissa on samankaltaista ideaa taustalla.

Listing 7.3. Lisää where:n käyttöä

```
ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 fab fbc = fac
    where
        fac jokua = fbc (fab jokua)

ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 fab fbc = tulos
    where
        tulos = \a ->fbc (fab a)

ex3 :: (a -> b) -> (c,a) -> b
ex3 fab (jokuc, jokua) = jokub
    where
        jokub = fab (jokua)

ex3 :: (a -> b) -> (c,a) -> b
ex3 fab cs= tulos
    where
        tulos = fab (snd cs)
```

Myös “case’ ja “if” lausekkeet esiintyivät vastauksissa, mutteivat yhtä usein kuin “where”. Näistä esimerkit alla 7.4. On myös mahdollista, että vastaajat ovat yrittäneet tehdä funktioista merkityksellisiä. Jos pelkän arvon palauttaminen on tuntunut oudolta funktiolta, niin on yritetty lisätä logiikkaa funktiolle. Täten funktio on selkeämmin laskenut jotain pelkän arvon palautuksen sijaan.

Listing 7.4. case ja if lausekkeiden käyttöä

```
ex5 :: Bool -> Bool
ex5 x = case x of
        True ->  True
```

```
        False -> False

ex6 :: Bool -> Bool -> Ordering
ex6 a b= if
    a == b
    then EQ
    else undefined
```

Kumpikaan näistä mainituista ei ollut erityisen yleistä, mutta pistivät kuitenkin silmään aineistosta. Nämä olivat osana yleisempää havaintoa liittyen tarpeettomien rakenteiden käyttöön ja haasteisiin syntaksin kanssa. Osa oppilaista käytti tarpeettomia rakenteita, kuten edellä mainitut lambdat, where ja case -lausekkeet. Havaintoon kuuluvat myös yleiset ongelmat oikean syntaksin kanssa. Useilla oppilailla oli haasteita palauttaa toimivalla syntaksilla olevia ratkaisuja ja tämä osin nosti joillain palautusten määrää, kun ratkaisua iteroitiin vahvasti.

Tyyppien osalta vaikeuksia aiheuttivat “Either”-tyypin käyttö ja etenkin sisäkkäinen “Either” oli selkeästi osalle vaikea saada oikein. Eniten haasteita aiheutti kuitenkin tyyppimääritelmät, joissa toteutettava funktio otti parametrina funktion. Funktiotyyppi oli siis haastava hahmottaa ja ymmärtää osalle. Etenkin sen käyttö vaati osalta oppilaista useita palautuksia, jotta meni oikein. palautusten kommenteista päätellen tehtäviä joissa funktioita oli parametreina käytiin myös eniten selvittämässä ohjauksissa. Ohjaukset ovat paikallisopetusta kurssilla, jossa voi tehdä tehtäviä ja saada ohjaajilta apua sekä vinkkejä. Funktiotyyppeihin liittyen oli näistä selkeästi kommentteja osassa palautuksista. Muutenkin parametrien tunnistaminen funktioksi ja niiden oikea käyttö oli selkeästi haasteellisinta opiskelijoille palautusten perusteella. Tästä esimerkkejä alla 7.5 ja myös ylempänä 7.1 esitellyt lambdoja koskevat liittyvät myös väärinkäsityksiin funktioista.

Listing 7.5. Haasteita funktioiden kanssa

```
ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 f(x y) f(d e) = f(x e)

ex6 :: Bool -> Bool -> Ordering
ex6 = (True -> True) -> EQ
```

```
-- tehtävän määrittelemä tyyppi:
ex5 :: (a, b) -> (a -> b -> c) -> (a,c)

-- erilaisia vastauksia:
ex5 f = \x -> \y -> (x, (f(x y)))

ex5 (a,b) f = (a, f(f(f)))

ex5 (x, y) = \(Left x) -> \(Right y) -> z

ex5 (a,b)= ((\a -> (\c -> (\b -> (a,c)))))
```

Alla 7.6 vielä kaksi esimerkkiä “Either”-tyyppiä sisältävän tehtävän väärinkäsityksistä.

Listing 7.6. Haasteita Eitherin kanssa

```
ex4 = Either (Left ()) (Left True)
ex4 = Right (Right ()) (Right EQ)

ex4 = Either (Left ()) (Left True)
ex4 = Right (Left) (Right True EQ)
```

Monissa vastauksissa näkyi myös väärinymmärrykset sulkujen merkityksestä. Tämä johtaa hankaluuksiin ja väärinkäsityksiin etenkin funktiotyyppien kohdalla. Ongelma näkyy myös yllä listatuissa poiminnoissa palautuksista. Seuraavaksi tarkastellaan tehtävässä kerättyä palautetta.

Tehtävässä kerättiin palautetta ensimmäisen ja toisen vaiheen jälkeen. Palaute koostui kahdesta kysymyksestä Likert-asteikolla (1-7) ja vapaasta tekstikentästä, johon pystyi antamaan kysymyksiä ja kommentteja tehtävästä. Ensimmäinen kysymys oli “Kuinka selkeä tämä tehtävä oli?”, jossa suurempi luku tarkoitti selkeämpää ja pienempi hämmentävää. Toinen kysymys oli “Kuinka vaikea tämä tehtävä oli?”, jossa suurempi luku tarkoittaa vaikeaa ja pienempi helppoa. Tehtävän ensimmäisen osan kyselyssä kaikkien vastanneiden keskiarvo selkeydestä oli 3 ja vaikeudesta 3. Toisessa osassa keskiarvot olivat selkeydestä 4 ja

vaikeudesta 3. Eli näiden perusteella toinen tehtävä oli selkeämpi. Seuraavassa taulukossa 11 on avattu keskiarvoja tarkemmin ryhmittämällä palautuksien lukumäärän mukaan. Palautuksista näkyy kuinka monta kertaa oppilas on palauttanut tehtävän. Kohdat "Selkeys 1 ka." ja "Vaikeus 1 ka." tarkoittavat tehtävän ensimmäisen kohdan kyselyn tuloksien keskiarvoa ryhmälle, jolla on ollut sama määrä palautuksia. Tehtävän toisen osan kyselylle näkyy taulukossa vastaavat tulokset. Viimeisenä taulukossa näkyy vastaajien määrä, joka tarkoittaa kyseisen ryhmän kokoa. Eli kuinka moni oppilaista vastasi yhtä monta kertaa tehtävään.

Palautuksia	Selkeys 1 ka.	Vaikeus 1 ka.	Selkeys 2 ka.	Vaikeus 2 ka.	Vastaajia
2	3.5	3.5	3.5	3.5	22
3	3.8	3.9	3.8	3.9	19
4	3.0	4.1	3.0	4.1	15
5	3.6	3.4	3.6	3.4	10
6	4.0	3.4	4.0	3.4	8
7	3.8	4.0	3.8	4.0	9
8	3.0	4.4	3.0	4.4	8
9	3.0	3.0	3.0	3.0	1
10	3.5	3.2	3.5	3.2	4
11	4.2	3.5	4.2	3.5	4
13	4.3	3.7	4.3	3.7	3
14	4.0	3.5	4.0	3.5	2
15	3.0	4.5	3.0	4.5	2
16	3.0	3.0	3.0	3.0	1
17	4.7	4.0	4.7	4.0	3
18	3.8	2.8	3.8	2.8	4
19	3.0	3.0	3.0	3.0	2
20	4.8	1.8	4.8	1.8	4
23	4.0	5.0	4.0	5.0	2
24	5.0	2.0	5.0	2.0	1
25	3.0	7.0	3.0	7.0	1
26	3.7	3.7	3.7	3.7	3
30	4.0	7.0	4.0	7.0	1
31	2.0	2.0	2.0	2.0	1

34	4.0	5.0	4.0	5.0	1
36	4.0	4.0	4.0	4.0	1
40	3.0	3.0	3.0	3.0	1
53	2.0	3.0	2.0	3.0	1
55	5.0	3.0	5.0	3.0	1
100	4.0	3.0	4.0	3.0	1
115	2.0	3.0	2.0	3.0	1
129	3.0	3.0	3.0	3.0	1
133	4.0	3.0	4.0	3.0	1

Taulukko 11: Ohjelmointitehtävän kyselyn tuloksia

Taulukosta 11 nähdään käsityksen selkeydestä ja vaikeudesta vaihtelevan palautuksista huolimatta. Tämä selittyy todennäköisesti sillä, että oppilailla on erilaisia vastaustyylejä. Osa oppilaista palauttaa tiehään tahtiin ja osa haluaa miettiä ratkaisunsa valmiiksi kerralla. Kuitenkin seitsemää yleisintä palautusmäärää tarkastellessa voidaan havaita käsityksen vaikeudesta kasvavan molemmissa tehtävän osissa kun palautusten määrä kasvaa. Käsitykset selkeydestä eivät seuraa vaikeutta samoja taulukon rivejä tarkastellessa.

Ohjelmointitehtävän palautuksissa esiintyi paljon samoja ongelmia kuin, mitä ensimmäisessäkin mahdollisten tyyppien lukumäärän päättely -tehtävässäkin nousi esiin. Esimerkiksi “Either”-tyyppiin liittyvät virheet esiintyivät siten, että oli mahdollisesti ajateltu Eitheriä untagged unionina eli “Left (Left True)” sijaan oli vastattu vain “Left True”. Vastauksissa oli myös monia muita virheitä “Either”-tyyppiin liittyen, joissa osassa jopa esiintyi se itse esim. “Either Left ...”. Myös sulut olivat usein näiden kohdalla väärin ja aiheuttivat virheitä ja hämmennystä.

7.3 Rajoitteet

Alaluvussa esitetään tutkimukseen liittyvät rajoitteet. Rajoitteita on useita ja niistä ehkä huomattavin on otannan suhteellinen pieni koko. Tutkielman aineisto on yhdeltä kurssil-

ta ja yhdeltä vuodelta, joten tämä jo rajoittaa mahdollisen aineiston määrää huomattavasti. Kurssilta myös valittiin vain kaksi tehtävää, joka on hyvin pieni osuus kokonaistehtävämäärästä. Toisaalta tutkielma keskittyi tyyppijärjestelmään liittyviin haasteisiin, joten sen osalta valitut tehtävät olivat oleellisimpia. Aiempina vuosina oppilasmäärät ovat olleet pienempiä, koska kurssi on ollut vapaaehtoinen, mutta viimeisimpänä vuonna 2017, jolta tutkielman aineistokin on oli kurssi pakollinen ja siten oppilasmäärä suurempi. Suuremman oppilasmäärän lisäksi otanta on laajempi, koska pakollisena kurssina sille osallistuu laajempi kirjo erilaisia oppilaita. Osa oppilaista saattaa palauttaa tehtäviä sen kummempia ajattelematta ja iteroida paljon, joka johtaa keskiarvon vääristymiseen. Toisaalta nämä luvut usein kertovat enemmänkin siitä kuinka palautusjärjestelmää käytetään ja hyödynnetään kuin itse tehtävän haastavuudesta. Tämä huomio koskee etenkin ohjelmointitehtäviä, jotka suurin osa oppilaista tekee kurssin verkkosivulle upotetulla ohjelmointiympäristöllä.

Kysymysten asettelu voi myös vaikuttaa huomattavasti tehtävän haastavuuteen ja siten tutkielman tuloksiin. Etenkin ensimmäisessä arvioidussa tehtävässä epäselvä kysymys yhdessä alakohdassa voi nostaa sen vaikeimmaksi ja siten sen kysymyksen aiheen haastavimmaksi kohdaksi opiskelijoille. Tämän tutkielman tulokset ovat melko yhdenmukaiset ja edellä mainittu rajoite ei todennäköisesti vaikuttanut niihin. Toisen tehtävän analysoinnissa oli rajoitteena ohjelmointivastausten raaka muoto ja aeimpaa tutkimusta onkin tehty ohjelmointiympäristön palautuksen aikana tapahtuvan tyyppitarkastuksen virheitä tutkimalla. Näistä virheistä saa helpommin tuotettua selkeitä lukuja ja tilastoja.

Tällä samalla kurssilla on aiemmin tutkittu vaikeuksia ja väärinkäsityksiä modernin tyyppijärjestelmän kanssa (Tirronen 2014), sekä aloittelijoiden virheitä Haskell-ohjelmointikieleen liittyen (Tirronen, Uusi-Mäkelä ja Isomöttönen 2015). Näissä tutkimuksissa käytettiin juurikin edellä mainittua tapaa ja tutkittiin kääntäjän virheitä, jolloin tilastointi on helpompaa.

8 Tulokset ja jatkotutkimus

Luvussa esitellään havaitut tulokset ja verrataan niitä mahdollisiin oletuksiin. Esitellään myös jatkotutkimuksen mahdollisuudet.

Tässä tutkielmassa tarkasteltiin funktio-ohjelmoinnin kurssille osallistuneiden opiskelijoiden haasteita Haskell-ohjelmointikielen tyyppijärjestelmän kanssa. Kyseisellä kurssilla on aiemmin tutkittu vaikeuksia ja väärinkäsityksiä modernin tyyppijärjestelmän kanssa (Tirronen 2014), sekä aloittelijoiden virheitä Haskell-ohjelmointikieleen liittyen (Tirronen, Uusi-Mäkelä ja Isomöttönen 2015). Tämä tutkielma laajensi tutkimusta liittyen aloittelijoiden haasteisiin funktio-ohjelmoinnissa ja tarkemmin Haskell-ohjelmointikielessä keskittymällä kielen tyyppijärjestelmään.

Tavoitteena oli tunnistaa opiskelijoiden yleisimmät virheet, väärinymmärrykset ja haasteet tyyppijärjestelmän kanssa analysoimalla tehtävien palautuksia ja automaattitehtävien keräämiä lokitiedostoja. Kurssilla aineistoa kerättiin tallentamalla kaikki tehtävien palautukset ja niitä voidaan tarkastella jälkeenpäin. Osana tutkielmaa kurssille tehtiin myös lisää automaattitehtäviä, jotta saatiin kerättyä enemmän aineistoa liittyen tyyppijärjestelmään. Alkuperäinen tutkimuskysymys “Mitkä ovat opiskelijoiden suurimpia haasteita Haskellin tyyppijärjestelmän kanssa?” ei ollut tarpeeksi tarkka, joten sitä tarkennettiin kohdistumaan kahteen valittuun kurssitehtävään. Tarkennettuja kysymyksiä olivat seuraavat: “Onko valitusta tyyppien ymmärrystä testaavasta tehtävästä havaittavissa selkeitä haasteita?” ja “Löytyvätkö samat havaitut haasteet myös toisen valitun tehtävän tuloksista?”. Näiden kysymysten avulla tutkimuksen tavoite tarkentui ja sen oli mahdollista vastata paremmin tutkimuskysymyksiin.

Kahden valitun tehtävän analyysin pohjalta suurin haaste liittyen Haskell-ohjelmointikielen tyyppijärjestelmään oli funktiotyyppien ymmärtäminen ja käyttäminen. Ensimmäisessä tehtävässä funktiotyyppi nousi esiin kahdessa tilastollisesti vaikeimmalta vaikuttavassa tehtävässä. Myöskin toisen tehtävän tuloksia analysoidessa kävi ilmi funktiotyyppien olevan tehtävän vaikein konsepti. Oppilailla oli haasteita ymmärtää funktiotyypin käyttö funktion parametrina ja etenkin niiden käyttö funktion toteutusta ohjelmoidessa. Haasteet funktiotyyppien kanssa toistuivat siis toisessakin valitussa tehtävässä ja suurin haaste oli tehtävissä

sama.

Muita mainittavia haasteita olivat summa- ja tulotyyppien hahmottaminen ja sulkujen käyttö, sekä funktioaplikaatio. Summa- ja tulotyyppeihin liittyvät väärinkäsitykset ilmenivät etenkin ensimmäisessä tehtävässä, jossa pääteltiin mahdollisten arvojen lukumääriä annetuille tyypeille. Ohjelmointitehtävässä tämä esiintyi ongelmina etenkin “Either”-tyypin kanssa. Sulutus ja funktioaplikaatio menivät monilla sekaisin ohjelmointitehtävässä. Väärinymmärrykset sulutukseen liittyen tukevat virheitä funktioaplikaatiossa ja yleisesti funktiotyyppejä käsiteltäessä.

Tutkielman tulokset vahvistavat aiempia havaintoja Haskell-ohjelmointikieleen ja tyyppeihin liittyvistä haasteista. Kiinnostavaa onkin voidaanko näiden tulosten perusteella päätellä miten nykyistä opetusta voisi kehittää. Ainakin se on selvää, että funktiotyyppeihin, funktioaplikaatioon ja sulkuihin tulee kiinnittää enemmän huomiota. Myöskin summa- ja tulotyypeistä ja niiden käytöstä voisi olla lisää yksinkertaisia esimerkkejä, jotta niiden käyttöön ei jäisi epäselvyyksiä. Yksi mahdollinen kehityksen kohde voisi myös olla vaikeuksissa olevien oppilaiden auttaminen siten, että väärinkäsitykset saataisiin korjattua mahdollisimman aikaisin. Jos oppilaalle jää väärinkäsityksiä ja aukkoja ymmärrykseen esimerkiksi syntaksiin ja sulkuihin liittyen niin tämä vahvistaa ongelmia myös esimerkiksi funktiotyyppien kanssa. Tätä tosin on vaikea arvioida tämän tutkielman tulosten pohjalta. Kaiken kaikkiaan tutkielman tulokset pohjautuvat lopulta vain kahteen kurssin tehtävään, joten kovin tarkkoja parannuksia niistä on vaikea saada irti. Tulokset kuitenkin tukivat aiemminkin oppilaille vaikeiksi havaittuja teemoja, joten näihin olisi hyvä panostaa jatkossa mahdollisuuksien mukaan vieläkin enemmän.

Jatkotutkimuksena voisi tutkia löytyykö muilta vuosilta vastaavia tuloksia näiden tehtävien osalta. Aiempia ja tulevia vuosia voi olla mielenkiintoista verrata näiden havaintojen vahvistamiseksi. Jatkossa tosin ohjelmointitehtävissä kannattaa keskittyä aineistoon, jota on helpompi analysoida tilastollisesti. Esimerkiksi siis tyyppejä käsittelevien ohjelmointitehtävien kääntäjän antamat virheet voisivat olla hyvä lähtökohta.

Lähteet

Alasuutari, Pertti. 1999. “Laadullinen tutkimus kolmas uudistettu painos”. *Laadullinen tutkimus. Vastapaino, Tampere.*

Brown, Neil CC, ja Amjad Altadmri. 2014. “Investigating novice programming mistakes: Educator beliefs vs. student data”. Teoksessa *Proceedings of the tenth annual conference on International computing education research,* 43–50. ACM.

Brusilovsky, Peter, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola ym. 2014. “Increasing Adoption of Smart Learning Content for Computer Science Education”. Teoksessa *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference,* 31–57. ITiCSE-WGR ’14. Uppsala, Sweden: ACM. ISBN: 978-1-4503-3406-8. doi:10.1145/2713609.2713611. http://doi.acm.org/10.1145/2713609.2713611.

Cardelli, Luca. 1996. “Type Systems”. *ACM Comput. Surv.* (New York, NY, USA) 28, numero 1 (): 263–264. ISSN: 0360-0300. doi:10.1145/234313.234418. http://doi.acm.org/10.1145/234313.234418.

Chakravarty, Manuel M. T., ja Gabriele Keller. 2004. “The risks and benefits of teaching purely functional programming in first year”. *Journal of Functional Programming* 14 (01): 113–123. ISSN: 1469-7653. doi:10.1017/S0956796803004805. http://journals.cambridge.org/article_S0956796803004805.

Church, Alonzo. 1940. “A formulation of the simple theory of types”. *The journal of symbolic logic* 5 (2): 56–68.

Clack, Chris, ja Colin Myers. 1995. “The dys-functional student”. Teoksessa *Funtional Programming Languages in Education,* 289–309. Springer.

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt ja Shriram Krishnamurthi. 2004. “The structure and interpretation of the computer science curriculum”. *Journal of Functional Programming* 14 (04): 365–378.

Findler, Robert Bruce, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi ja Matthias Felleisen. 1997. “DrScheme: A Pedagogic Programming Environment for Scheme”. Teoksessa *Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education,* 369–388. PLILP ’97. London, UK, UK: Springer-Verlag. ISBN: 3-540-63398-7. http://dl.acm.org/citation.cfm?id=646452.692958.

Hall, Cordelia V, Kevin Hammond, Simon L Peyton Jones ja Philip L Wadler. 1996. “Type classes in Haskell”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18 (2): 109–138.

Hanenberg, Stefan, Sebastian Kleinschmager, Romain Robbes, Éric Tanter ja Andreas Stefik. 2014. “An empirical study on the impact of static typing on software maintainability”. *Empirical Software Engineering* 19, numero 5 (): 1335–1382. ISSN: 1573-7616. doi:10.1007/s10664-013-9289-1. https://doi.org/10.1007/s10664-013-9289-1.

Harper, Robert. 2016. *Practical Foundations for Programming Languages.* 2nd. New York, NY, USA: Cambridge University Press. ISBN: 1107150302, 9781107150300.

Heeren, Bastiaan, Daan Leijen ja Arjan van IJzendoorn. 2003. “Helium, for learning Haskell”. Teoksessa *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell,* 62–71. ACM.

Hutton, Graham. 2007. *Programming in Haskell.* Cambridge University Press.

Joosten, Stef, Klaas Van Den Berg ja Gerrit Van Der Hoeven. 1993. “Teaching functional programming to first-year students”. *Journal of Functional Programming* 3 (01): 49–65.

Järvinen, Pertti, ja Annikki Järvinen. 2004. *Tutkimustyön metodeista.*

Karavirta, Ville, Ari Korhonen ja Lauri Malmi. 2005. “Different learners need different resubmission policies in automatic assessment systems”. Teoksessa *Proceedings of the 5th Annual Finnish/Baltic Sea Conference on Computer Science Education,* 95–102.

Kleinschmager, S., R. Robbes, A. Stefik, S. Hanenberg ja E. Tanter. 2012. “Do static type systems improve the maintainability of software systems? An empirical study”. Teoksessa *2012 20th IEEE International Conference on Program Comprehension (ICPC),* 153–162. doi:10.1109/ICPC.2012.6240483.

Lipovaca, Miran. 2011. *Learn you a haskell for great good!: a beginner's guide.* no starch press.

Malmi, L., ja A. Korhonen. 2004. "Automatic feedback and resubmissions as learning aid". Teoksessa *IEEE International Conference on Advanced Learning Technologies, 2004. Proceedings.* 186–190. doi:10.1109/ICALT.2004.1357400.

Malmi, Lauri, Ari Korhonen ja Riku Saikkonen. 2002. "Experiences in Automatic Assessment on Mass Courses and Issues for Designing Virtual Courses". *SIGCSE Bull.* (New York, NY, USA) 34, numero 3 (): 55–59. ISSN: 0097-8418. doi:10.1145/637610.544433. http://doi.acm.org/10.1145/637610.544433.

Pierce, Benjamin C. 2002. *Types and programming languages.* MIT press.

Ruehr, Fritz. 2008. "Tips on Teaching Types and Functions". Teoksessa *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education,* 79–90. FDPE '08. Victoria, BC, Canada: ACM. ISBN: 978-1-60558-068-5. doi:10.1145/1411260.1411272. http://doi.acm.org/10.1145/1411260.1411272.

Saikkonen, Riku, Lauri Malmi ja Ari Korhonen. 2001. "Fully Automatic Assessment of Programming Exercises". *SIGCSE Bull.* (New York, NY, USA) 33, numero 3 (): 133–136. ISSN: 0097-8418. doi:10.1145/507758.377666. http://doi.acm.org/10.1145/507758.377666.

Tirronen, Ville. 2014. "Study on Difficulties and Misconceptions with Modern Type Systems". Teoksessa *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education,* 303–308. ITiCSE '14. Uppsala, Sweden: ACM. ISBN: 978-1-4503-2833-3. doi:10.1145/2591708.2591726. http://doi.acm.org/10.1145/2591708.2591726.

Tirronen, Ville, ja Ville Isomöttönen. 2012. "On the Design of Effective Learning Materials for Supporting Self-directed Learning of Programming". Teoksessa *Proceedings of the 12th Koli Calling International Conference on Computing Education Research,* 74–82. Koli Calling '12. Koli, Finland: ACM. ISBN: 978-1-4503-1795-5. doi:10.1145/2401796.2401805. http://doi.acm.org/10.1145/2401796.2401805.

Tirronen, Ville, Samuel Uusi-Mäkelä ja Ville Isomöttönen. 2015. “Understanding beginners’ mistakes with Haskell”. *Journal of Functional Programming* 25. ISSN: 1469-7653. doi:10.1017/S0956796815000179. http://journals.cambridge.org/article_S0956796815000179.

Yin, Robert K. 2003. “Case study research design and methods third edition”. *Applied social research methods series* 5.

Liitteet