Sampo Osmonen

Funktionaalinen paradigma ohjelmoijan näkökulmasta

Tietotekniikan kandidaatintutkielma

25. tammikuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Sampo Osmonen

Yhteystiedot: sampo.e.osmonen@student.jyu.fi

Työn nimi: Funktionaalinen paradigma ohjelmoijan näkökulmasta

Title in English: Functional paradigm from the programmer's point of view

Työ: Kandidaatintutkielma

Sivumäärä: 29+0

Tiivistelmä: Funktionaalinen ohjelmointiparadigma on ohjelmointityyli, joka tarjoaa kiinnostavan vaihtoehdon suositulle imperatiiviselle paradigmalle. Tässä tutkielmassa luodaan katsaus funktionaaliseen paradigmaan vertailemalla sitä imperatiiviseen paradigmaan eri näkökulmista. Samalla esitellään tarkemmin funktionaalisen paradigman tärkeitä ominaisuuksia. Osoittautuu, että funktionaalinen paradigma voi usein olla kilpailukykyinen vaihtoehto sovelluskehitykseen. Lisäksi todetaan, että funktionaalinen paradigma tarjoaa ohjelmointiin monia hyödyllisiä työkaluja, joita imperatiivinen paradigma ei tue. Näiden työkalujen käyttöönotto ja integrointi imperatiivisiin ohjelmointikieliin voi tarjota useita etuja ohjelmistojen kehittämiseen.

Avainsanat: ohjelmointi, funktionaalinen, imperatiivinen, paradigma

Abstract: Functional programming paradigm is a style of writing programs, one that offers an interesting alternative to the popular imperative paradigm. In this thesis an overlook on the functional paradigm is taken by comparing it with the imperative paradigm. Consequently several important characteristics of the functional paradigm are introduced. It turns out that the functional paradigm can often be a competitive option for software development. Additionally, it's established that the functional paradigm offers many useful tools the imperative paradigm doesn't support. Deploying these techniques and integrating them into existing languages is likely to offer several advantages for software development.

Keywords: programming, functional, imperative, paradigm

Sisältö

1 Johdanto

Kaikki tietokoneohjelmat muodostuvat jollain ohjelmointikielellä kirjoitetusta ohjelmakoodista. Ohjelmaa on mahdollista kuvata paitsi kohdetietokoneen suoraan tukemalla ohjelmointikielellä (yleisnimitykseltään *assembly*) myös niin kutsutuilla *korkeamman tason* ohjelmointikielillä. Korkeamman tason kielet abstrahoivat tietokoneen käyttämän assembly-kielen toiminnan omien semantiikkojensa taakse, ja mahdollistavat ohjelmien kirjoittamisen assemblysta poikkeavilla tyyleillä. Korkemman tason kielten ansiosta samalla tavoin toimivan ohjelman koodin voi esittää hyvin monenlaisessa eri muodossa, kunhan työkalut koodin kääntämiseen tietokoneen ymmärtämään muotoon ovat saatavilla. Tapaa esittää ohjelmakoodi muotoiltuna tiettyjen tyylisääntöjen mukaisesti kutsutaan *ohjelmointiparadigmaksi*.

Tietokoneiden kehityksen aikana on syntynyt monia erilaisia ohjelmointiparadigmoja. Paradigmat voivat tarjota ohjelmoijalle erilaisia etuja, ja siksi niiden välillä esiintyy kilpailua. Hallitsevassa asemassa ovat pitkään olleet *imperatiivinen paradigma* ja sen aliparadigmat (TIOBE Software 2015). Valtaosa nykyisin yleisessä käytössä olevista ohjelmointikielistä noudattaa imperatiivista paradigmaa (TIOBE Software 2015). Eräs toinen paradigma on *funktionaalinen paradigma*. Vaikka funktionaalisen paradigman suosio ei ylläkään imperatiivisen tasolle, on funktionaalisuus yllättävän tehokas tyyli jäsentää ohjelmia. Jotkin tahot ovat osoittaneet kiinnostustaan funktionaaliseen ohjelmointiin (Marlow 2015), ja toistaiseksi funktionaalisissa kielissä on erityisen kiinnostavaa niiden lupaama potentiaali ohjelmoinnin tehostamiseen.

Luvussa 2 esitellään funktionaalisen ja imperatiivisen paradigman eroja käsitteistön ja konseptien tasolla erityisesti lausekkeiden ja lausekkeiden käytön sekä sivuvaikutusten ja niiden puutteen näkökulmista. Lisäksi tarkastellaan lyhyesti paradigmojen käytännönläheisempiä eroja niiden tarjoamien työkalujen muodossa.

Luvussa 3 käsitellään etuja, joita funktionaaliset kielet imperatiivista paradigmaa noudattaviin kieliin nähden voivat tarjota. Lisäksi luodaan katsaus ongelmiin, joita

funktionaalisilla kielillä imperatiivisiin kieliin nähden usein on.

Luvussa 4 tarkastellaan objektiivisesti mitattavia eroja, joita eräiden funktionaalisten ja imperatiivisten kielten välillä on tutkimuksissa todettu. Pohditaan myös paradigmojen keskinäisiä heikkouksia ja vahvuuksia sovelluskehitysprojektissa näiden tietojen perusteella.

2 Funktionaalisuuden ja imperatiivisuuden eroja

Funktionaalisen ja imperatiivisen paradigman lähestymistavat ohjelmointiin ovat luonteeltaan hyvin erilaisia. Imperatiivinen ohjelmakoodi kuvaa sarjaa vaiheittaisia käskyjä suoritettavaksi ohjelman ajon aikana. Funktionaalinen paradigma voidaan mieltää osaksi laajempaa *deklaratiivista paradigmaa*, jota noudattava ohjelmakoodi kuvaa käskyjen sijaan valmista ohjelmaa. Usein korostetaan, että funktionaalinen koodi kuvaa, *mitä* ohjelma on, imperatiivinen koodi kuvaa, *miten* ohjelma jotain tekee.

2.1 Lausekkeet ja lauseet

Imperatiivisen koodin tärkeitä rakenneosia ovat *lauseet* (Hudak 1989, ss. 361). Lauseet edustavat käskyjä, joita ohjelman suorituksesta vastaava (abstrakti) kone toteuttaa annetussa järjestyksessä. Näiden käskyjen tarkoituksena on muutella suoritettavan ohjelman ”sisäistä” implisiittistä tilaa niin, että edetään tilasta toiseen käskyjen määräämällä tavalla (Hudak 1989, ss. 361). Tietokoneohjelma voi tyypillisesti myös ohjailla fyysistä isäntäkonettaan tai vastaanottaa siltä syötteitä ohjelmoijan määräämien lauseiden perusteella. Tavallisesti imperatiivisessa kielessä ohjelman sisäistä tilaa kuvataan muuttujien ja ohjelman suoritusvaiheen avulla (Backus 1978, ss. 615). Lauseiden avulla voidaan hallita muuttujien arvoja eri vaiheissa suorituksen edetessä tai hallita suorituksen kulkua (Hudak 1989, ss. 361). Näin ohjelman myöhempi suoritus voi riippua sen aiemmasta tilasta, suoritusvaiheesta ja muuttujista. Ohjelmasta saadaan tilaa hallitsemalla rakennettua järkevä kokonaisuus.

Funktionaalista paradigmaa noudattavassa koodissa lauseita ei ole tai niiden käyttöä on rajattu. Ohjelma kirjoitetaan sen sijaan deklaratiivisesti käyttämällä *lausekkeita* (Hudak 1989, ss. 361). Lauseista poiketen lausekkeet eivät kuvaa käskyjä (Hudak 1989, ss. 361), vaan itse valmista ohjelmaa tai sen osia. Jokaiseen validiin lausekkeeseen liittyy jokin semanttinen merkitys, arvo (Hudak 1989, ss. 361). Lausekkeita voidaan yhdistellä ohjelmointikielen asettamien sääntöjen mukaisesti, ja näin lausek-

keiden yhdistelmänä voidaan kuvata koko toivottu ohjelma. Kirjoittajan kokemusten mukaan funktionaalisissa kielissä lausekkeet yleensä kuvaavat erityisesti vakioarvoja, muuttujia, funktioita sekä funktioiden applikaatioita. Näitä rakenneosia koostetaan suuremmiksi kuvauksiksi, ohjelmiksi. Kuten matematiikassa, lauseke ei usein esiinny sievennetyimmässä muodossaan, vaan sen tarkan arvon selvittäminen vaatii lausekkeen evaluointia (Vegdahl 1984, ss. 1051). Tästä lausekkeiden evaluoinnista muodostuu ohjelman suoritus (Hudak 1989, ss. 360).

2.2 Sivuvaikutukset

Niin kutsutuilla puhtailla deklaratiivisilla lausekkeilla ei ole *sivuvaikutuksia* — kykyä aiheuttaa tilan muutoksia suoritettavan ohjelman sisällä tai antaa suoria käskyjä isäntäkoneen toteutettavaksi (Hudak 1989, ss. 362). Tämä on kirjoittajan näkemyksen mukaisesti mahdollisesti funktionaalisuuden ja imperatiivisuuden merkitsevin ero ohjelmoijan kannalta. Sivuvaikutusten puutteen vuoksi funktionaalisessa ohjelmakoodissa ole käsitettä tietyssä järjestyksessä muuteltavasta ohjelman tilasta (Hudak 1989, ss. 361), ja siksi funktionaalisen koodin ei tarvitse eksplisiittisesti ottaa kantaa ohjelman suoritusjärjestykseen, vaan tämä vastuu siirtyy ohjelmoijalta koodia suorittavalle alustalle (Hudak 1989, ss. 398). Imperatiivista paradigmaa käytettäessä ohjelmoija joutuu ohjaamaan suoritusta myös suoritusjärjestyksen suhteen.

Ilman kykyä minkäänlaiseen sivuvaikutusten käyttöön funktionaaliset ohjelmointikielet olisivat hyvin rajoittuneita. Lähes poikkeuksetta ohjelman odotetaan voivan myös hallita konetta, jossa se suoritetaan. Myös funktionaalisiin kieliin vaaditaan siis jonkinlainen mekanismi tiettyjen sivuvaikutusten ilmaisemiseksi. Jotkin osin funktionaaliset kielet ratkaisevat tämän ongelman sallimalla lausekkeiden lisäksi vapaan lauseiden käytön (Hudak 1989, ss. 362). Sen sijaan toiset funktionaaliset kielet eristävät koneen ohjaamiseen tarkoitetut rakenteet selkeästi muusta ohjelmakoodista omiin osiinsa. Jälkimmäisiä kieliä, jotka ”normaalissa” ohjelmakoodissaan sallivat ainoastaan sivuvaikutuksettomien lausekkeiden käytön, kutsutaan puhtaiksi funktionaalisiksi ohjelmointikieliksi.

2.3 Moniparadigmakielet

Suurin osa ohjelmointikielistä ei noudata puhtaasti pelkästään yhtä paradigmaa. Näitä kieliä kutsutaan moniparadigmakieliksi. Funktionaalisuutta painottavat kielet, jotka sallivat myös lauseiden käytön, ovat osin funktionaalisia moniparadigmakieliä. Toisaalta on olemassa myös imperatiivisuutta painottavia moniparadigmakieliä, jotka sisältävät useita funktionaalisia mekanismeja mutta silti kannustavat ohjemoijaa käyttämään imperatiivisia lauseita. Lisäksi puhtaasti imperatiivisiksi mielletyt kielet sallivat lausekkeiden käytön tiettyjen lauseiden osina (Backus 1978, ss. 616) ja usein myös sisältävät joitakin pohjimmiltaan funktionaalisiksi tulkittavia ominaisuuksia, kuten funktioina toimivat aliohjelmat tai rekursio. Selkeimmät erot imperatiivisen ja funktionaalisen paradigman välillä ovat havaittavissa mahdollisimman puhtaasti yhtä paradigmaa edustavien kielten välillä. Siksi jatkossa tässä tutkielmassa termillä funktionaalinen kieli viitataan oletusarvoisesti kieleen, joka noudattaa funktionaalista paradigmaa puhtaasti eikä salli suoria sivuvaikutuksia aiheuttavien rakenteiden käyttöä vapaasti tavallisen koodin seassa. Termillä imperatiivinen kieli viitataan kieleen, joka tukee mahdollisesti useitakin funktionaalisia ominaisuuksia, mutta korostaa imperatiivisuutta, lauseiden käyttöä ja sivuvaikutuksia.

2.4 Funktionaalisia ja imperatiivisia työkaluja

Imperatiivisiin kieliin tottuneelle puhdas tai lähes puhdas funktionaalinen kieli saattaa aluksi tuntua rajoittavalta. Osin tämä saattaa johtua siitä, että paradigmaa vaihtaessaan ohjelmoija joutuu mukautumaan erilaisiin uusiin ajattelumalleihin ja työkaluihin. Toisaalta aiemmin todetusti funktionaalisten kielten deklaratiivisuus aiheuttaa merkittävän rajoitteen imperatiivisiin kieliin nähden: funktionaaliset kielet eivät salli lauseiden käyttöä tai rajoittavat sitä huomattavasti. Nykyaikaiset imperatiiviset kielet sen sijaan mahdollistavat sekä lauseiden että lausekkeiden käytön hyvin monipuolisesti. Lisäksi jotkin imperatiiviset kielet sisältävät monia funktionaalisten kielten kielten eduiksi yleisesti katsottuja ominaisuuksia.

Lauseiden ja sivuvaikutusten puute tarkoittaa, että kaikki imperatiivisissa kielissä käytettävät työkalut eivät ole sellaisinaan mahdollisia funktionaalisissa kielissä. Merkittäviä funktionaalisista kielistä puuttuvia imperatiivisia ominaisuuksia ovat muuteltavat muuttujat, datatietueiden muokkaaminen, kyky kontrolloida ohjelman suoritusvaihetta käskyillä sekä kyky kutsua isäntäkoneen toimintoja tavallisen koodin seassa. Funktionaaliset kielet kuitenkin tarjoavat omia vaihtoehtoisia työkaluja ohjelmoijan käyttöön niin, että samat ongelmat ovat ratkaistavissa molempia paradigmoja edustavilla kielillä. Niin funktionaalinen kuin imperatiivinenkin paradigma mahdollistavat Turing-täydellisten kielten luomisen ja siten kaikkien tietokoneella laskettavien laskujen suorittamisen (Hudak 1989).

Sivuvaikutusten puutteesta väistämättä seuraa, että muuttujat, joita puhtaassa funktionaalisessa koodissa esiintyy, eivät voi vaihtaa arvoaan ohjelman suorituksen aikana muuttaen suorituksen sisäistä tilaa. Muuttujat muistuttavat käytännössä enemmän siis imperatiivisten ohjelmakielten vakioita. Näin ollen muuttujien manipulointi ohjelman tilan hallitsemiseen suorituksen aikana ei puhtaissa funktionaalisissa kielissä ole mahdollista. Vastaavat laskut voidaan funktionaalisissa kielissä kuitenkin esittää kuvaamalla muokattavaa tilaa eksplisiittisesti jonain kielen arvona (Hudak 1989, ss. 405) ja manipuloimalla sitä funktioilla tavallisen arvon tavoin.

Muuttujien tavoin kaikki data on sivuvaikutusten puutteen vuoksi puhtaissa funktionaalisessa kielessä muuttumatonta. Tämä tarkoittaa, että mitään olemassaolevaa oliota ei ole mahdollista päivittää destruktiivisesti siten, että se alkuperäisessä sijainnissaan muuttuisi (Hudak 1989; Okasaki 1999, ss. 405**)**. Sen sijaan uuden olion luominen vanhan olion arvoja hyödyntämällä on mahdollista, jolloin alkuperäinen olio säilyy ennallaan, ja uusi olio voidaan muodostaa halutusti alkuperäisen perusteella. Funktionaalisissa kielissä olion muokkaaminen luo implisiittisesti uuden, alkuperäisestä tietueesta transformoidun tietueen ja jättää alkuperäisen olion ennalleen edelleen käytettäväksi (Okasaki 1999). Myös olioiden toistuva muokkaaminen on funktionaalisessa kielessä mahdollista esittää ketjuttamalla funktioita, jotka palauttavat muokatun version oliosta osana paluuarvoaan.

Koska funktionaalisissa kielissä ei ole käsitettä ohjelman suoritusvaiheesta, ei tois-

torakenteiden kuvaaminen imperatiivisten kielten tavoin siirtymäkäskyjä käyttäen ole mahdollista. Iterointi on imperatiivisissa kielissä useimmiten primitiivinä tarjottu toiminto. Sen sijaan puhtaissa funktionaalisissa kielissä iteroinnin korvaa rekursio (Hudak 1989, ss. 361). Rekursiota käytetään funktioiden määritelmissä niin, että funktio määrittelee itsensä osin omaan määritelmäänsä perustuen. Rekursiivisilla funktioilla, kuten rekursiolla yleisesti, on mahdollista ilmaista mielivaltaisen kokoisia laskuja rajatun kokoisilla lausekkeilla. Näin rekursion avulla voidaan suoraan kuvata kompaktisti jopa äärettömän pituisia laskuja, ja siten tarvetta varsinaiseen vaiheittaiseen iterointiin laskun saamiseksi ei ole. Lähes kaikki imperatiivisetkin kielet tukevat rekursiota, mutta puhtaissa funktionaalisissa kielissä rekursio on varsinaisten iterointirakenteiden puutteen vuoksi erityisessä asemassa.

Funktionaalisissa kielissä sivuvaikutusten mallintaminen ei ole yhtä suoraviivaista kuin imperatiivisissa kielissä. Koska funktionaalisten kielten kielioppiin ei sisälly käsitettä sivuvaikutuksista, ohjelmaa suorittavan tietokoneen ohjaaminen ja implisiittisen tilan käsittely vaativat uusia lähestymistapoja. Funktionaalisiin kieliin onkin kehitetty malleja, jotka pystyvät kuvaamaan sivuvaikutuksia ilman, että koodin deklaratiivisuudesta joudutaan luopumaan. Suosittu menetelmä tähän on *monadien* käyttö. Seuraava kuvaus monadeista on yksinkertainen tulkinta sivustolta HaskellWiki 2015. Monadi on datatyyppi, joka liittää jonkin tyyppiseeen arvoon lisätietoa, eräänlaisen kontekstin, ja jonka alkioiden avulla voidaan kuvata peräkkäisiä laskuja. Monadeja voidaan ketjuttaa niin, että aikaisemman monadin arvo syötetään funktiolle, joka palauttaa uuden monadin jollain arvolla ja kontekstilla. Aikaisemman monadin ja funktion palauttaman monadin kontekstit yhdistetään implisiittisesti. Lopullinen saatava monadi sisältää funktion palauttaman monadin arvon kontekstissa, joka on saatu alkuperäisen monadin ja funktion palauttaman monadin kontekstit jotenkin yhdistämällä. Monadien avulla ”epäpuhtaat” sivuvaikutukset, kuten käskyt tietokoneelle, ovat näin abstrahoitavissa monadien konteksteihin niin, että ne eivät ole ohjelmoijan suoraan hallittavissa tai nähtävissä ohjelmakoodissa, vaan ne käsitellään automaattisesti suorituksen aikana. Ketjumaisen rakenteensa ansiosta peräkkäiset monadit kuvaavat myös toimintojen suoritusjärjestystä. Monadit ovat hyvä tapa kuvata sivuvaikutuksia funktionaalisessa koodissa.

3 Funktionaalisuuden etuja ja haittoja

Funktionaaliseen ohjelmointiin liittyy joitakin hyödyllisiä ominaisuuksia imperatiivisuuteen verrattuna. Vaikka nykyiset imperatiivisuutta korostava moniparadigmakielet pystyvätkin toteuttamaan suuren osan näistä konsepteista, funktionaaliset kielet usein kannustavat ohjelmoijaa paremmin hyödyntämään funktionaalisen tyylin etuja (Hudak 1989, ss. 361). Ominaisuuksia, jotka ovat erityisesti funktionaaliselle paradigmalle keskeisiä, ovat muun muassa ensiluokkaiset funktiot (engl. first-class function), funktionaaliset tietorakenteet, laiska evaluointi (engl. lazy evaluation) ja viitteellinen avoimuus (Vegdahl 1984; Okasaki 1999; Hudak 1989). Funktionaaliseen paradigmaan liittyy myös omat ongelmansa, joita tarkastellaan luvun viimeisessä kappaleessa.

3.1 Ensiluokkaiset funktiot

Funktionaalisissa kielissä poikkeuksetta funktiot ovat niin sanotusti *ensiluokkaisia*, eli ne ovat tavallisia arvoja siinä missä muidenkin datatyyppien arvot (Narayan, Gopinath ja Sridhar 2008). Imperatiivisessa paradigmassa proseduurit tai funktiot eivät voi toimia tavallisten arvojen tavoin, sillä ne käsitetään ainoastaan muistiin ladatuksi ohjelmakoodiksi. Funktioiden ensiluokkaisuus näkyy käytännössä siten, että funktioita voidaan ainakin tallettaa muuttujiin, antaa parametreina toisille funktioille ja palauttaa toisten funktioiden tuloksina (Narayan, Gopinath ja Sridhar 2008). Kyseisistä ominaisuuksista käytetään myös termiä *korkeamman asteen funktiot* (engl. higher-order functions) (Narayan, Gopinath ja Sridhar 2008). Korkeamman asteen funktiot tukevat modulaarisemman ja yleiskäyttöisemmän koodin kirjoittamista (Hughes 1989) — esimerkiksi toiminnallisuuden vastaanottaminen parametrina funktioon usein lisää funktion käyttökohteiden määrää. Imperatiiviset kielet voivat jäljitellä korkeamman asteen funktioita esimerkiksi käyttämällä osoittimia funktioihin.

Muita mahdollisia ensiluokkaisten funktioiden ominaisuuksia ovat sisäkkäiset funktiot, anonyymit funktiot, sulkeumat (eng. closure) sekä osittainen applikointi . Useat

funktionaaliset kielet tukevat yhtä tai useampaa näistä toiminnallisuuksista, kun taas imperatiivisissa kielissä ne ovat harvinaisempia.

Sisäkkäiset funktiot mahdollistavat uusien funktioiden määrittämisen toisten funktioiden näkyvyysalueen sisällä (Willcock ym. 2006). Tällöin sisempi funktio saa käyttöönsä kaikki ulomman funktionsa saatavilla olevat muuttujat (Willcock ym. 2006). Funktioiden julistaminen sisäkkäin mahdollistaa funktioiden näkyvyyden tarkemman hallinnan, sillä toisen funktion näkyvyysalueen sisällä määritetty funktio ei näy globaalisti. Tämän ansiosta voidaan välttää täyttämästä globaalia näkyvyysaluetta apufunktioilla, joille vain yhdellä funktiolla on käyttöä. Funktioiden määrittely sisäkkäin voi myös pienentää sisäfunktioiden käyttämää parametrijoukkoa, sillä ulomman funktion alueelta löytyvät muuttujat voidaan jättää kirjoittamatta eksplisiittisinä parametreina.

Lambdafunktiot ovat nimettömiä funktioita — funktioita, joille ei määrätä eksplisiittistä nimeä (Willcock ym. 2006). Tuki lamdafunktioiden luonnille tekee kertaluontoisten funktioiden määrittämistä syntaktisesti vaivattomampaa ja siistimpää (Willcock ym. 2006).

Sulkeumat mahdollistavat sisäkkäisten funktioiden ulkofunktioltaan saamien muuttujien käytön myös, kun sisäfunktio poistuu vanhempansa näkyvyysalueelta (Willcock ym. 2006). Käytännössä tämä tarkoittaa, että jos sisäfunktio käyttää vanhempansa muuttujia, nämä muuttujat ovat edelleen saatavilla, kun funktio palautetaan vanhemman paluuarvona ja sitä kutsutaan muualta. Sulkeumien mahdollistama kyky säilyttää niiden alkuperäisellä näkyvyysalueella olevaa dataa on erittäin hyödyllinen ominaisuus, ja se voi parantaa koodin modulaarisuutta — esimerkiksi voidaan luoda funktioita, jotka palauttavat uusia, käyttäjän antamien parametrien perusteella räätälöidysti toimivia funktioita.

Osittaisen applikoinnin avulla funktion parametrit voidaan syöttää osissa niin, että osa parametreista jätetään pois (HaskellWiki 2007). Jokaisen osittaisen applikaation tuloksena saadaan uusi funktio, joka ottaa vastaan alkuperäisen funktion loput parametrit (HaskellWiki 2007). Osittaista applikointia voidaan käyttää esimerkiksi, jos

osa aiemmin määritetyn funktion parametreista halutaan kiinnittää tiettyyn arvoon niin, että se voidaan antaa parametriksi jollekin funktiolle, joka odottaa pienemmän määrän parametreja vastaanottavaa funktiota.

3.2 Funktionaaliset tietorakenteet

Koska kaikki puhdas funktionaalinen data on muuttumatonta kuten luvussa 2 todettiin, funktionaalisten tietorakenteiden käsittely eroaa imperatiivisista muokautuvista tietorakenteista. Funktionaaliset tietorakenteet ovat pysyviä (engl. persistent), eli niitä käsiteltäessä kaikki aikaisemmat versiot ovat uusimman lisäksi aina tarpeen mukaan saatavilla alkuperäisissä sijainneissaan (Okasaki 1999). Kontrastina imperatiivisissa kielissä yleensä käytetään niin kutsuttuja hetkellisiä tietorakenteita, jotka muokattaessa korvaavat alkuperäisen version muokkauksen tuloksella, jolloin viimeisimmän muokkauksen tuloksena saatu versio on ainoa saatavilla oleva versio tietorakenteesta (Okasaki 1999).

Datan muuttumattomuus myös helpottaa jakamista funktionaalisissa kielissä, koska mitä tahansa pysyviä tietorakenteita voidaan liikutella ilman pelkoa korruptiosta alkuperäisessä sijainnissa. Näin ollen dataa ja tietorakenteita voidaan turvallisesti jakaa koodin eri osien tai suoritettavan ohjelman eri säikeiden välillä. Tämä datan jakamisen ongelmattomuus voi osaltaan helpottaa funktionaalisen ohjelman rinnakkaistamista imperatiiviseen ohjelmaan nähden.

Funktionaaliset datarakenteet ovat luonteeltaan monimutkaisempia kuin muokattavat vastineensa, sillä niiltä odotetaan enemmän toiminnallisuutta pysyvyyden muodossa (Okasaki 1999). Siksi myös niiden toteuttaminen on usein työläämpää, ja tehottoman toteutuksen tapauksessa niiden suorituskyky voi olla jopa asymptoottisesti huonompi kuin imperatiivisten vastineidensa (Okasaki 1999). Toisaalta oikeita optimointistrategioita hyödyntämällä on useimmiten mahdollista toteuttaa funktionaalisia tietorakenteita, jotka ovat asymptoottisesti yhtä tehokkaita kuin niiden hetkelliset vastineet (Okasaki 1999).

3.3 Ilmaisuvoimaiset tyyppijärjestelmät

Tehokkaat staattiset tyyppijärjestelmät eivät ole funktionaalisuuteen suoraan liittyvä ominaisuus, mutta useat funktionaaliset tai osin funktionaaliset kielet, kuten Haskell, SML ja F#, tarjoavat toisiaan muistuttavat monipuoliset staattiset tyyppijärjestelmät.

Algebralliset datatyypit tarkoittavat tyyppejä, joihin voi kuulua useita toisistaan erillisiä arvoja (jolloin kyseessä on summatyyppi), ja joiden arvot voivat sisältää arvoja muista tyypeistä (jolloin kyseessä on tulotyyppi) (Harper 2012, luvut 11, 12). Algebrallista tyyppiä edustavaa arvoa ja sen sisältämiä muita arvoja voidaan tarkastella ja liittää muuttujiin menetelmällä, jonka englannin kielinen nimi on *pattern matching* (Harper 2012, ss. 105). Pattern matching -menetelmää voidaan käyttää myös edellyttämään arvoilta tiettyä rakennetta esimerkiksi funktioiden parametreissa ja valitsemaan tämän perusteella haluttu vartalo funktiolle (Hudak 1989, ss. 388). Algebraliset datatyypit mahdollistavat monipuolisten tyyppien käsittelyn turvallisesti kevyellä syntaksilla (Schrage, IJzendoorn ja Gaag 2005). Algebrallisen tyyppijärjestelmän on esitetty myös pystyvän vahvistamaan ohjelman oikeellisuuden paremmin kuin yleisemmin käytössä olevat imperatiivisten kielten käyttämät tyyppijärjestelmät (Schrage, IJzendoorn ja Gaag 2005), mikä vastaa kirjoittajan kokemuksia. Kirjoitushetkellä algebralliset datatyypit ovat saatavilla myös joissakin imperatiivisuutta tukevissa kielissä, esimerkkinä Rust (The Rust Project Developers 2015, luvut 7, 8).

Tyyppien päättely (engl. type inference) on ominaisuus, jonka ansiosta kääntäjä voi päätellä koodissa esiintyvien lausekkeiden tyyppejä automaattisesti niiden käytön perusteella ilman, että ohjelmoijan tarvitsee eksplisiittisesti niitä ilmoittaa (Hudak 1989, ss. 375–376). Kääntäjä pyrkii mahdollisuuksien mukaan johtamaan lausekkeiden tyypit ennestään tunnettujen tyyppien perusteella. Näin voidaan vähentää tyyppi-ilmoitusten käyttöä koodissa ja silti säilyttää tyyppitarkastusten tuomat edut. Tyyppi-ilmoitusten vähentäminen helpottaa kirjoittamista, tekee koodista lyhyempää ja mahdollisesti helpommin luettavaa, joskin viimeinen kohta on osin subjektiivinen ominaisuus. Tyyppien päättely on saatavilla joissakin imperatiivisissa kielis-

sä.

3.4 Laiska suoritus

Laiska suoritus (engl. lazy evaluation) on koodin suoritusstrategia, jossa koodissa kuvattuja laskuja ei suoriteta ennen kuin niiden tuloksia suorituksen aikana tarvitaan (Harper 2012). Jotta koodin osa voitaisiin suorittaa laiskasti, ei se voi sisältää sivuvaikutuksia. Tämä johtuu siitä, että laiskaa suoritusstrategiaa käytettäessä koodin osien suoritusjärjestystä ei voida ennen ajoa määrätä, jolloin sivuvaikutukset voisivat tapahtua satunnaisessa järjestyksessä ja olla siksi hyödyttömiä käytännöllisten sovellusten kannalta. Laiska suoritus on oletusarvoisena suoritusstrategiana käytössä joissakin puhtaissa funktionaalisissa kielissä, mutta sitä voidaan hyödyntää myös moniparadigmakielten deklaratiivisissa osissa, joskin tämä on usein hankalaa (Hudak 1989, ss. 362). Imperatiiviset kielet käyttävät yleensä pääsääntöisesti datapohjaista suoritusta (engl. eager evaluation) eli ahnetta suoritusta. Ahne suoritus tarkoittaa, että ohjelmakoodin kuvaamat laskut lasketaan ohjelmaa suoritettaessa sheti kun tarvittava data on saatavilla. Ahneeseen suoritukseen nähden laiska suoritus vaatii nykyisillä tietokonearkkitehtuureilla ylimääräistä kirjanpitoa toistaiseksi laskemattomista arvoista ja siten vähentää hieman suorituskykyä (Vegdahl 1984; Hudak 1989, ss. 1056**)**, mutta toisaalta laiskalla suorituksella on joitakin tärkeitä etuja (Hudak 1989, ss. 383–384).

Eräs laiskan suorituksen etu on, että sen käyttö voi vähentää tarpeettomien laskujen suorittamista ohjelmaa ajettaessa (Vegdahl 1984, ss. 1056). Tämä perustuu siihen, että ohjelmakoodissa voidaan usein määrittää laskuja, joiden tuloksia ei välttämättä ohjelman suorituksessa aina tarvita riippuen ohjelman saamista dynaamisista syötteistä. Ahnetta suoritusta käytettäessä laskun tulos on selvitettävä heti kun mahdollista niin, että tulos on saatavilla, jos muut koodin osat tarvitsevat sitä. Ahne suoritus voi tämän vuoksi aiheuttaa laskun tarpeettoman suorittamisen, jos tulosta ei myöhemmässä suorituksessa käytetäkään missään. Laiskaa suoritusta käytettäessä sen sijaan lasku suoritetaan vasta, kun huomataan sen tulokselle olevan käyttöä, ja siten voidaan mahdollisesti parantaa suorituskykyä dynaamisesti valikoimalla

koodista vain tarpeelliset osat suoritettaviksi (Vegdahl 1984; ss. 1056 Hudak 1989, ss. 384–385).

Laiskaa suoritusta käytettäessä lasketut tulokset talletetaan implisiittisesti väliaikaiseen tietorakenteeseen, josta ne ovat nopeasti löydettävissä myöhemmillä käyttökerroilla (Hudak 1989; Harper 2012, ss. 384–385, 400–401). Tekniikka on englannin kieliseltä nimeltään *memoization*. Memoization-tekniikka voi joissain laskuissa vähentää suoritusaikaa merkittävästi pientä muistitilan käytön kasvua vastaan ilman, että ohjelmoijan tarvitsee itse huolehtia laskujen tulosten säilömisestä tai käyttää monimutkaisempia algoritmeja. Toisaalta kuitenkin toistuva memoization voi aiheuttaa merkitseviä haittoja suorituskykyyn suoritusta odottavien laskujen kasautuessa (Vegdahl 1984).

Laiska suoritus mahdollistaa äärettömien tietorakenteiden käsittelyn ilman erityisiä toimenpiteitä ohjelmoijalta (Hudak 1989, ss. 385–386). Koska tietorakenteesta käsitellään vain tarvittavat osat, ei tietorakennetta luoda yhtenä monoliittisena operaationa, vaan sitä muodostetaan koodissa annetun määritelmän perusteella vaiheittain sitä mukaa kuin sen osia toisaalla käytetään (Okasaki 1999). Äärettömän tietorakenteen käsittely ei siis mahdollisesti aiheuta loputonta silmukkaa kuten ahneesti suoritettaessa kävisi. Äärettömien tietorakenteiden siirtely ja läpikäynti on laiskan suorituksen ansiosta huomattavan yksinkertaista ja tehokasta, kun taas ahnetta suoritusta käytettäessä jouduttaisiin äärettömien datamäärien mallintamiseen käyttämään erillisiä rakenteita äärettömän datan käsittelyyn. Äärettömät datarakenteet kuten linkitetyt listat voivat yksinkertaistaa koodia huomattavasti, ja ovat hyödyllisiä esimerkiksi fysikaalisessa laskennassa (Karczmarczuk 1999).

Laiskan suorituksen eduksi voidaan laskea myös ohjelman strukturointi luonnollisemmiksi modulaarisiksi osiksi. Hughes 1989 esittää, että laiska suoritus mahdollistaa ohjelmien ja niiden osien erottelun *generaattoreihin*, jotka tuottavat dataa, ja *selektoreihin*, jotka käsittelevät generaattoreiden tuottamaa dataa. Selektorit voivat käyttää generaattoreiden dataa omien tarpeidensa mukaisesti ilman, että itse generaattoriin tarvitsee liittää logiikkaa tuotettavan datan määrän rajaamiseksi. Tämä parantaa komponenttien modulaarisuutta.

3.5 Rinnakkaislaskenta ja viitteellinen avoimuus

Puhdas funktionaalinen koodi on deklaratiivista, ja siten ei ota kantaa järjestykseen, jossa ohjelman osat suoritetaan. Vastuu sopivan suoritusjärjestyksen valinnasta on alustalla, jolla ohjelmaa suoritetaan, siis ohjelman kääntäjällä tai tulkilla (Hudak 1989, ss. 398). Tämän vuoksi suoritusalustalla on hyvin vapaat kädet optimoida ohjelman suoritusta niin, että sen osia suoritetaan automaattisesti rinnakkaisesti (Hudak 1989, ss. 398). Ohjelmoijalta voidaan vaatia vihjeitä siitä, mitä osia ohjelmasta rinnakkaistetaan, mutta koodin semanttista rakennetta ohjelmoijan ei tarvitse muuttaa mitenkään. Automaattinen rinnakkaistaminen on ohjelmoijan kannalta vaivaton ratkaisu ongelmaan, joka imperatiivisissa kielissä useimmiten vaatii ohjelman rakenteen osittaista muuttamista (Vegdahl 1984, ss. 1051).

Viitteellinen avoimuus (engl. referential transparency) tarkoittaa, että kielen lausekkeet ovat vapaasti korvattavissa samoja arvoja vastaavilla lausekkeilla minkä tahansa lausekkeen sisällä ja milloin tahansa ilman, että vastaavan lausekkeen arvo muuttuu (Hudak 1989, ss. 362). Puhdas funktionaalinen koodi on deklaratiivisuudestaan johtuen viitteellisesti avointa (Hudak 1989, ss. 362). Käytännössä tämä ohjelmoijan näkökulmasta tarkoittaa, että muuttujien arvot ovat vakioita kuten kappaleessa 2 todettiin, ja että funktiot ovat puhtaita ja palauttavat aina saman tuloksen tiettyjä parametreja kohden.

Ohjelmoijan kannalta viitteellinen avoimuus on hyödyllinen ominaisuus ohjelmien oikeellisuutta todistettaessa (Vegdahl 1984). Sen ansiosta ei ole tarvetta pitää kirjaa mistään ohjelman sisäisestä tilasta tai suorituksen historiasta, vaan voidaan vapaasti käsitellä yksittäisiä lausekkeita korvaamalla muuttujia niiden arvoilla, korvaamalla lausekkeita muuttujilla, ja suorittamalla ohjelmointikielen primitiivisiä toimenpiteitä kuten esimerkiksi yhteenlaskuja. Lausekkeiden käsittely vastaa siis hyvin läheisesti matemaattisten lausekkeiden käsittelyä.

Lisäksi viitteellinen avoimuus voi myös jossain määrin helpottaa ohjelmakoodin lukemista, sillä koodin osa-lausekkeet ovat aina ymmärrettävissä ilman tietoa suuremmasta kontekstista (Schrage, IJzendoorn ja Gaag 2005), kun taas imperatiivises-

sa koodissa esiintyvät arvot saattavat suorituskontekstista riippuen muuttua. Loogisesti voidaan myös todeta, että viitteellinen avoimuus voi osaltaan helpottaa ohjelman osien testaamista (Vegdahl 1984, ss. 1052–1053), sillä esimerkiksi ohjelman puhtaat funktiot palauttavat aina saman tuloksen tiettyä syötettä kohden riippumatta mistään ulkoisista tekijöistä.

3.6 Funktionaalisuuden ongelmia

Vaikka funktionaalisuuteen liittyykin monia etuja, on erilaisilla funktionaalisuutta painottavilla kielillä myös omat ongelmapuolensa. Näistä heikkouksista on hyvä olla tietoinen ennen funktionaalisen kielen käyttöönottoa projektissa.

Eräs funktionaalisten kielten ongelma ovat rajoitukset, joita ne asettavat ohjelmoijalle suosittuihin imperatiivisiin kieliin verrattuna. Kuten luvussa 2 kuvattiin, puhtaista funktionaalisista kielistä puuttuu monia hyödyllisiä imperatiivisia ominaisuuksia. On esitetty, että funktionaalisten kielten tarjoamat vaihtoehtoiset ratkaisut eivät välttämättä aina ole yhtä suoraviivaisia käyttää kuin imperatiiviset ratkaisut (Vegdahl 1984, ss. 1052). Esimerkiksi sivuvaikutusten puute saattaa monimutkaistaa tilan välittämistä ohjelman osien välillä, sillä tila pitää eksplisiittisesti välittää eri osiin sen sijaan, että päivitys yhdessä paikassa riittäisi päivittämään jaetun datan. On myös esitetty väitteitä funktionaalisen ja imperatiivisen paradigman keskinäisestä lähestyttävyydestä ohjelmoijille. Esimerkiksi Eric Lippert uskoo, että imperatiivinen tyyli on lähempänä useimpien luontaisia ajattemalleja (Lippert 2010). Vaikka lähestyttävyys onkin vaikeasti mitattava ja osin subjektiivinen ominaisuus, joka tapauksessa funktionaalisten ja imperatiivisten kielten erilaisuus väistämättä vaatii imperatiivisia kieliä käyttäneeltä ohjelmoijalta perehtymistä uuteen paradigmaan.

Usein korostetaan funktionaalisten kielten ongelmia suorituskyvyssä (Vegdahl 1984, ss. 1053–1054). Funktionaalisten kielten semantiikat vastaavat useimpiin imperatiivisiin kieliin verrattuna heikommin lähes kaikkien nykyisten tietokoneiden käyttämän von Neumann -arkkitehtuurin rakenteita (Vegdahl 1984; Backus 1978, ss. 1053).

Tämä asettaa luonnollisesti haasteita funktionaalisen koodin kääntämiseen tehokkaaksi assembly-koodiksi. Kirjoittajan näkemyksen mukaan kielen heikompi vastaavuus arkkitehtuuriin saattaa myös joissain tapauksissa vähentää ohjelmoijan mahdollisuuksia optimoida koodia kyseiselle alustalle. Luvussa 4 tarkastellaan lähemmin funktionaalisten ja imperatiivisten kielten suorituskykyä kirjoitushetkellä.

4 Kielten kvalitatiivisia eroja

Vaikka eri ohjelmointikielten ominaisuuksia voidaan analysoida ja vertailla helposti, ei ole mitään suoraviivaista tapaa verrata ohjelmointikieliä objektiivisesti. Kielten kvalitatiivisten ominaisuuksien empiirisellä vertailulla voidaan kuitenki saada suuntaa antavia tuloksia kielten välisestä paremmuusjärjestyksestä. Tällaisia tutkimuksia on tehty useita. Nämä tutkimukset vertailevat kielten paremmuuden indikaattoreina muun muassa suorituskykyä, ohjelmointiin kuluvaa aikaa, debuggaamiseen kuluvaa aikaa, ajon aikana esiintyvien virheiden määrää tai koodin kokonaismäärää.

4.1 Suorituskyky

Ohjelmointikielen valinnan vaikutus ohjelman suorituskykyyn vaihtelee suuresti ohjelmista ja toteutuksista riippuen. Siksi eri kielten ja paradigmojen asettaminen yleiseen paremmuusjärjestykseen suorituskyvyn perusteella on hankalaa.

Nanz ja Furia 2015 kirjoittavat suorittamastaan vertailusta Rosetta Code -sivuston tarjoamien ohjelmien kvalitatiivisista ominaisuuksista. Tutkimus antaa mahdollisuuden analysoida funktionaalisen ja imperatiivisen paradigman eroja rajoitetusti. Vertailun ainoa puhdas funktionaalinen kieli on käännettävä kieli Haskell, ja lisäksi vertailussa on mukana tavukoodina tulkittava moniparadigmakieli F#, joka painottaa funktionaalisuutta. Muut kielet ovat ensisijaisesti imperatiivisia, ja niistä natiiviksi assemblyksi tai tavukoodiksi käännettäviä ovat C, C#, Go ja Java.

Tutkimuksen tulokset osoittavat, että suoritettujen laskennallisesti vaativien ohjelmien keskiarvoisen suoritusajan perusteella funktionaalista paradigmaa edustavan Haskell-kielen suorituskyky on heikompi kuin kaikkien imperatiivisten kielten. Tutkimuksen tulkinnan mukaisesti C ja Go olivat suurella erolla tehokkaampia kuin Haskell, Java keskitasoisella erolla ja C# pienellä erolla. Toisaalta myös Javan ja C#:n suorituskyky erosi selvästi nopeimman C-kielen suorituskyvystä. Tutkimuksessa verrattiin lisäksi suorituskykyä tehtävissä, jotka eivät vaatineet raskaita laskusuo-

rituksia, vaan edustivat ”tavallista” algoritmillisesti yksinkertaisissa ohjelmistoissa käytettävää koodia. Näissä tehtävissä Haskell oli yllättäen lähellä nopeimpien C- ja Go-kielten tasoa. Toinen mukana ollut funktionaalisuutta korostava kieli oli F#. Sekin jäi keskimäärin jälkeen imperatiivisista kielistä, mutta oli suorituskykyisempi kuin Haskell. F# käyttää moniparadigmakielenä Haskellia vapaammin imperatiivisia rakenteita, mikä saattoi olla tekijänä sen parempaan tulokseen. F# oli lähellä C#:a, hitainta imperatiivisista kieltä, mikä on ymmärrettävää, sillä molemmat käännettiin testissä samanlaiseksi tavukoodiksi samalle virtuaalikoneelle.

Muistia Haskell ja F# käyttivät enimmillään enemmän kuin imperatiiviset kielet. Erot olivat kuitenkin samaa kokoluokkaa kuin erot muistia ahneemmin ja niukemmin käyttävien imperatiivisten kielten välillä.

Pankratius, Schmidt ja Garreton 2012 kirjoittavat vertailusta Java- ja Scala-kielten ominaisuuksista rinnakkaislaskentaa vaativassa ohjelmoinnissa. Java on imperatiivinen olio-ohjelmointia korostava kieli, Scala taas on moniparadigmakieli, joka toisaalta tukee imperatiivista ohjelmointia ja olio-ohjelmointia kattavasti. Suurin osa tutkimuksessa käytetyistä ohjelmista sisälsi sekä imperatiiviseksi että funktionaaliseksi tutkimusryhmän puolesta luokiteltua koodia. Tutkimuksen tuloksena todettiin, että funktionaalisten ominaisuuksien laaja käyttö ei vähentänyt ohjelmien suorituskykyä. Toisaalta parhaiten menestyi koodi, joka sisälsi vastaavat määrät imperatiivisia ja funktionaalisia osia. Tutkimus siis korostaa molempien paradigmojen hyödyntämisen etuja. Imperatiivisuutta välttävä koodi menestyi huonommin, ja funktionaalisten datarakenteiden ja niiden muokkaamisen todettiin aiheuttaneen testissä suorituskykyongelmia.

Tutkimuksessa, jonka Nanz ja Furia 2015 suorittivat, tehokkaiksi havaittuihin imperatiivisiin C- ja Go-kieliin nähden puhtaan funktionaalisen Haskell-kielen suorituskyky on selvästi heikompi, mutta toisaalta se on lähempänä toisten, vähemmän suorituskykyisten imperatiivisten kielten tasoa. Vastaavan suuruisia suorityskykyeroja esiintyy myös eri imperatiivisten kielten välillä. Moniparadigmakieli F# suoriutui tutkimuksessa Haskellia paremmin, ja tutkimuksen, jonka Pankratius, Schmidt ja Garreton 2012 suorittivat, perusteella funktionaalisten elementtien käyttäminen

voi olla suorituskyvyn kannalta jopa hyödyllistä. Tutkimusten tulosten perusteella on siis tärkeää valita harkiten käytettävä ohjelmointikieli, jos ohjelmalta vaaditaan mahdollisimman hyvää suorituskykyä. Keskimäärin puhtaiden funktionaalisten kielten suorituskyky vaikuttaa jäävän imperatiivisia heikommaksi, mutta toisaalta moniparadigmakielet hämärtävät rajoja. Sopivan kielen valinta ei myöskään rajoitu pelkästään funktionaalisten ja imperatiivisten kielten välille, vaan myös näiden ryhmien sisällä on merkitseviä eroja. Tärkeämpää on keskittyä vaadittavaan suorituskykyyn ja kielten eroihin kuin eri paradigmoihin. Mitä vähemmän ohjelma painottaa algoritmillista suorituskykyä, sitä vapaammin käytettävän kielen voi valita, ja myös puhtaat funktionaaliset kielet ovat varteenotettavia vaihtoehtoja.

4.2 Tuottavuus

Laajassa käytössä olevalle kielelle on ensiarvoisen tärkeä ominaisuus, että se tukee toimivien ohjelmistojen kehittämistä mahdollisimman nopeasti ja ongelmattomasti. Kehittämisnopeutta on eri ohjelmakielten välillä suorituskyvyn tavoin vaikea mitata tarkasti, mutta empiiriset tutkimukset voivat antaa hyvää yleiskuvaa kielten eroista.

Tutkimus, jonka Pankratius, Schmidt ja Garreton 2012 suorittivat, tarjoaa hyvän katsauksen funktionaalisten ja imperatiivisten menetelmien välisiin eroihin tuottavuudessa. Tutkimuksessa verrattiin Java- ja Scala-kieliä, jotka toimivat samalla virtuaalikoneella ja voivat käyttää samoja Javan standardikirjastoja, joten kielet tarjoavat otollisen pohjan paradigmojen vertailulle muiden muuttujien vähäisyyden ansiosta. Tutkimuksen mukaan vastaavien ohjelmien kirjoittaminen Scala-kielellä vaati kehittäjiltä enemmän aikaa Java-kieleen verrattuna. Erityisesti testaamiseen ja debuggaamiseen kului enemmän aikaa Scalaa käytettäessä. Selityksenä tälle esitettiin, että Scalan ominaisuuksien luonne teki debuggaamisesta vaikeampaa ohjelmoijan taidoista riippumatta. Rinnakkaistettujen ohjelmien tuottamisessa Scala ja Java olivat hyvin tasavertaisia ajan suhteen.

Scala ei moniparadigmakielenä täysin vastaa puhtaampia funktionaalisia kieliä, mut-

ta suuri osa funktionaalisista ominaisuuksista on siinä silti saatavilla. Vaikka tutkimuksessa käytetyt ohjelmat sisälsivätkin paljon imperatiivista koodia, antaa tutkimus silti suuntaa siihen, miten funktionaalisen koodin käyttöönotto projektissa vaikuttaa tuottavuuteen. Tutkimuksen pohjalta voidaan siis olettaa, että funktionaalisen koodin tuottaminen vie ohjelmoijalta yleisesti enemmän aikaa kuin imperatiivisen. On silti huomioitava, että tutkimuksessa tutkittiin vain kolmen eri ohjelman toteutuksia. Nämä ohjelmat eivät mitenkään voi edustaa kaikkia sovelluksia, joihin kieliä voidaan käyttää — eri kokoiset ja eri tarkoituksiin toteutettavat projektit voivat hyötyä eri paradigmoista. Scala-kielen debug-työkalujen puutteellisuuden aiheuttama lisäys työaikaan saattaa olla tulevaisuudessa pienennettävissä, sillä Scala ei ole yhtä vakiintunut kieli kuin vanhempi Java.

4.3 Bugit

Ohjelmointikielen valinnalla voi olla merkittävä vaikutus ajonaikaisten virheiden esiintyvyyteen. Erityisesti staattisesti tyypitetyissä kielissä esiintyy yleisesti vähemmän virheitä kuin dynaamisesti tyypitetyissä, sillä tyyppijärjestelmä löytää monet ohjelmoijan tekemät virheet jo kääntämisen aikana. Eroja on tutkittu myös funktionaalisten ja imperatiivisten ohjelmointikielten välillä.

Nanz ja Furia 2015 kirjoittavat, että näytteet, jotka oli toteutettu funktionaalisilla kielillä (Haskell ja F#) sisälsivät suhteellisen vähän ajonaikaisia bugeja. Imperatiivisista staattisesti tyypitetyistä kielistä Java- ja C-näytteet kaatuivat todennäköisemmin ajon aikana. Vahvan tyyppijärjestelmän ja rajoitetusti olio-ominaisuuksia sisältävän Go-kielen näytteet olivat kuitenkin selvästi bugittomimpia. Tämän tutkimuksen perusteella on vaikea tehdä johtopäätöksiä funktionaalisten ja imperatiivisten paradigmojen keskinäisestä bugien esiintymismäärästä.

Ray ym. 2014 kirjoittavat suorittamastaan tutkimuksesta, jossa eri ohjelmointikieliä edustavien projektien sisältämien bugien määriä vertailtiin GitHubin suosituimpien repositorioiden perusteella. Tutkimuksessa todettiin heikko suhde kielen funktionaalisuuden ja pienemmän bugien määrän välillä.

Paradigmojen vaikutukset bugien määrään ovat siis suhteellisen pieniä, mutta todennäköisesti funktionaalista paradigmaa käyttävät kielet sisältävät keskimäärin hieman vähemmän bugeja.

4.4 Muut ominaisuudet

Muita kiinnostavia ohjelmointikielten ominaisuuksia ovat muun muassa luettavuus ja vaadittava koodin määrä. Luettavuudesta ei juurikaan ole tehty tutkimuksia. Koodin määrää sen sijaan on verrattain helppo verrata — funktionaalinen koodi on useissa tutkimuksissa todettu keskimäärin tiiviimmäksi (Nanz ja Furia 2015; Pankratius, Schmidt ja Garreton 2012).

5 Yhteenveto

Tässä tutkielmassa tarkasteltiin aluksi funktionaalisen ja imperatiivisen ohjelmointiparadigman välisiä käsitteellisiä eroja, edettiin esittelemään funktionaalisuuden hyötyjä ja haittoja, ja lopuksi kuvattiin funktionaalisten ja imperatiivisten kielten kvalitatiivisia eroja. Näin tutkielma tarjoaa katsauksen funktionaalisen paradigman luonteeseen imperatiiviseen paradigmaan tutustuneelle sekä esittelee funktionaalisuuteen liittyviä ominaisuuksia, jotka voivat vaikuttaa valintaan funktionaalisen ja imperatiivisen kielen välillä ohjelmistoprojektissa.

Funktionaalinen paradigma osoittautui huomionarvoiseksi vaihtoehdoksi suositulle imperatiiviselle paradigmalle. Se sisältää monia hyödyllisiä ohjelmointia helpottavia ominaisuuksia, joita imperatiivisessa paradigmassa ei ole. Funktionaalisten kielten suorituskyky ja tuottavuus ovat vertailukelpoisella tasolla suosittuihin korkeamman tason imperatiivisiin kieliin nähden. Käyttötarkoituksesta riippuen funktionaalinen ohjelmointikieli saattaa ominaisuuksiensa suhteen olla kannattavampi valinta kuin imperatiivinen kieli. Tämän vuoksi on yllättävää, miten pieni funktionaalista paradigmaa suosivien kielten käyttöaste on verrattuna imperatiivisten kielten käyttöasteeseen.

Merkittävä tekijä funktionaalisten kielten vähäisessä käytössä on varmasti imperatiivisten kielten historiallinen johtoasema. Imperatiivinen paradigma ja sen aliparadigmat ovat vakiinnuttaneet johtavan asemansa ohjelmistotuotannossa, ja siten ne ovat saavuttaneet kehittäjäyhteisön laajan tuen. Tämä tuki voi kirjoittajan oletuksen mukaan vaikuttaa kielen valintaan ohjelmistoprojektissa. Funktionaalisilla kielillä on myös omat heikkoutensa imperatiivisiin kieliin nähden, eikä funktionaalisuus välttämättä tuo mukanaan pelkkiä etuja. Tämä projektista riippuen saattaa vähentää funktionaalisten kielten houkuttavuutta. Syitä funktionaalisen paradigman suhteellisen pienelle käyttöasteelle voi olla useita.

Ongelmistaan huolimatta funktionaalisten konseptien laajempi käyttöönotto olisi kirjoittajan näkemyksen luultavimmin hyödyllistä ohjelmistokehitykselle. Sopivien

ongelmien ratkaisuun funktionaalisuus soveltuu luontevasti, ja näin sen käyttö harkitusti voi lisätä ohjelmoijien tuottavuutta, parantaa ohjelmistojen luotettavuutta tai parantaa ohjelmakomponenttien modulaarisuutta. Rinnakkaislaskennan yleistyminen muun muassa moniytimisten prosessorien muodossa puolestaan saattaa korostaa ohjelman suoritusjärjestystä abstrahoivien funktionaalisten rakenteiden hyödyllisyyttä.

Funktionaalisilla kielillä on kuitenkin kiistatta rajoitteensa imperatiivisiin kieliin nähden, ja niiden soveltuvuus voi riippua ratkaistavan ongelman luonteesta. Siksi usein puhdas funktionaalinen kieli ei välttämättä ole käytännössä kannattavin ratkaisu, vaan paras valinta voi olla jokin moniparadigmakieli. Moniparadigmakielet hämärtävät rajoja paradigmavalintojen välillä ja korostavat yksittäisten kielten merkitystä. Sopiva moniparadigmakieli saattaa tarjota projektiin optimaalisen kompromissin funktionaalisten ja imperatiivisten ominaisuuksien väliltä. Moniparadigmakielten edullisuutta puoltaa, että monet nykyisistä suosituimmista ohjelmointikielistä, kuten Python ja JavaScript, voidaan luokitetella moniparadigmakieliksi, ja lisäksi monet aiemmin hyvin imperatiiviset kielet, kuten C++ ja Java, ovat omaksuneet funktionaalisia ominaisuuksia.

Kirjoittaja uskoo, että funktionaalisten kielten käyttöaste tulevaisuudessa nousee nykyisestä. Jos moniparadigmakielten suosion kasvu ja paradigmojen sekoittuminen jatkuvat, leviävät samalla väistämättä myös funktionaaliset konseptit laajemmalle. Tarkemmin funktionaalisen paradigman asemaa ohjelmistokehityksen tulevaisuudessa on kuitenkin erittäin vaikea ennustaa. Joka tapauksessa jo kirjoitushetkellä funktionaalinen paradigma tarjoaa ohjelmoijalle uusia työkaluja, jotka voivat tehostaa työskentelyä ja parantaa koodin laatua. Funktionaalisten käsitteiden omaksuminen voi monelle osoittautua kannattavaksi valinnaksi, ja se voi myös antaa uutta perspektiiviä imperatiiviseen ohjelmointiin.

Lähteet

Backus, John. 1978. “Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs”. *Commun. ACM* (New York, NY, USA) 21, numero 8 (elokuu): 613–641. ISSN: 0001-0782. doi:10.1145/359576.359579.

Harper, Robert. 2012. *Practical foundations for programming languages.* Cambridge University Press.

HaskellWiki. 2007. “Partial application”. Viitattu 13. joulukuuta 2015. https://wiki.haskell.org/index.php?title=Partial_application&oldid=14654.

———. 2015. “Monad”. Viitattu 26. lokakuuta 2015. https://wiki.haskell.org/index.php?title=Monad&oldid=59910.

Hudak, Paul. 1989. “Conception, Evolution, and Application of Functional Programming Languages”. *ACM Comput. Surv.* (New York, NY, USA) 21, numero 3 (syyskuu): 359–411. ISSN: 0360-0300. doi:10.1145/72551.72554.

Hughes, John. 1989. “Why functional programming matters”. *The computer journal* 32 (2): 98–107.

Karczmarczuk, L. 1999. “Scientific computation and functional programming”. *Computing in Science Engineering* 1, numero 3 (toukokuu): 64–72. ISSN: 1521-9615. doi:10.1109/5992.764217.

Lippert, Eric. 2010. “Why hasn’t functional programming taken over yet?” Viitattu 13. joulukuuta 2015. http://stackoverflow.com/revisions/2835936/2.

Marlow, Simon. 2015. “Fighting spam with Haskell”. Viitattu 9. joulukuuta 2015. https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/.

Nanz, S., ja C.A. Furia. 2015. "A Comparative Study of Programming Languages in Rosetta Code". Teoksessa *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, 1:778–788. Toukokuu. doi:10.1109/ICSE.2015.90.

Narayan, G., K. Gopinath ja V. Sridhar. 2008. "Structure and Interpretation of Computer Programs". Teoksessa *Theoretical Aspects of Software Engineering, 2008. TASE '08. 2nd IFIP/IEEE International Symposium on*, 73–80. Kesäkuu. doi:10.1109/TASE.2008.40.

Okasaki, Chris. 1999. *Purely functional data structures.* Cambridge University Press.

Pankratius, Victor, F. Schmidt ja G. Garreton. 2012. "Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java". Teoksessa *Software Engineering (ICSE), 2012 34th International Conference on*, 123–133. Kesäkuu. doi:10.1109/ICSE.2012.6227200.

Ray, Baishakhi, Daryl Posnett, Vladimir Filkov ja Premkumar Devanbu. 2014. "A Large Scale Study of Programming Languages and Code Quality in Github". Teoksessa *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 155–165. FSE 2014. Hong Kong, China: ACM. ISBN: 978-1-4503-3056-5. doi:10.1145/2635868.2635922.

Schrage, Martijn M., Arjan van IJzendoorn ja Linda C. van der Gaag. 2005. "Haskell Ready to Dazzle the Real World". Teoksessa *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, 17–26. Haskell '05. Tallinn, Estonia: ACM. ISBN: 1-59593-071-X. doi:10.1145/1088348.1088351.

The Rust Project Developers. 2015. "The Rust Reference". Viitattu 13. joulukuuta 2015. https://doc.rust-lang.org/nightly/reference.html.

TIOBE Software. 2015. "TIOBE Software: TIOBE Index". Viitattu 26. lokakuuta 2015. http://www.tiobe.com/.

Vegdahl, S.R. 1984. "A Survey of Proposed Architectures for the Execution of Functional Languages". *Computers, IEEE Transactions on* C-33, numero 12 (joulukuu): 1050–1071. ISSN: 0018-9340. doi:10.1109/TC.1984.1676387.

Willcock, Jeremiah, Jaakko Järvi, Doug Gregor, Bjarne Stroustrup ja Andrew Lumsdaine. 2006. “Lambda expressions and closures for C++”.