Tampereen yliopisto

Seppo Kolehmainen

FUNKTIONAALINEN OHJELMOINTI

Periaatteet, tekniikat ja hyödyt

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Huhtikuu 2021

TIIVISTELMÄ

Seppo Kolehmainen: Funktionaalinen ohjelmointi – Periaatteet, tekniikat ja hyödyt
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2021

Perinteisesti ohjelmointia ja sovelluskehitystä on lähestytty imperatiivisen ohjelmointiparadigman kautta. Funktionaalisen ohjelmointiparadigman suosio on kuitenkin alkanut kasvaa, ja siitä etsitään ratkaisuja ongelmiin, joita imperatiivinen paradigma ei ole onnistunut ratkaisemaan. Tässä tutkielmassa tarkastellaan funktionaaliselle ohjelmoinnille tyypillisiä ominaisuuksia ja tekniikoita, sekä tuodaan esille hyötyjä, joita funktionaalisella paradigmalla voidaan saavuttaa.

Tutkielma on muodoltaan kirjallisuuskatsaus. Sen aineisto on koottu hakemalla aiheeseen liittyviä artikkeleita tietojenkäsittelytieteeseen keskittyvistä julkaisutietokannoista. Artikkeleiden valinnassa on painotettu uudempia julkaisuja, mutta mukaan on otettu myös näkökulmaltaan teorialähtöisiä vanhempia artikkeleita.

Funktionaalisen ohjelmoinnin keskiössä on toiminnallisuuden toteuttaminen funktioita määrittelemällä. Funktio on ominaisuuksiltaan tiukasti määritelty ja se muistuttaakin läheisesti diskreetin matematiikan funktiota, joka määritellään usein kuvaukseksi lähtöjoukosta tulosjoukkoon. Funktion sivuvaikutuksella tarkoitetaan sitä, että funktio on parametreistaan johdetun arvon palauttamisen lisäksi vuorovaikutuksessa ympäristönsä kanssa. Funktionaalisessa ohjelmoinnissa lähtökohtana on, etteivät funktiot aiheuta sivuvaikutuksia lainkaan. Ohjelmien toiminnallisuuksien toteuttamiseksi sivuvaikutukset ovat kuitenkin välttämättömiä, joten funktionaaliset ohjelmointikielet sisältävät erilaisia tekniikoita niiden mahdollistamiseksi. Funktionaalisen ohjelmoinnin yksi keskeisimpiä piirteitä kuitenkin on, että sivuvaikutuksia pyritään välttämään ja silloin, kun ne ovat välttämättömiä, ne toteutetaan hallitusti.

Tutkielman aineistosta nousi esille useita merkittäviä hyötyjä, joita funktionaalisella ohjelmointiparadigmalla voi saavuttaa. Funktionaalinen ohjelmakoodi on ilmaisuvoimaista, helposti testattavaa, ja sen sisäisen toimintalogiikan tarkastelu on suoraviivaista. Funktionaalinen ohjelmointi tarjoaa tekniikoita ohjelmakoodin uudelleenkäyttöön, ja sen käytöllä on havaittu olevan yhteys sovelluskehittäjien tuottavuuteen. Myös rinnakkaislaskenta, jonka toteuttamista on perinteisesti pidetty haastavana, pysyy funktionaalisen ohjelmoinnin ominaisuuksien avulla helposti hallittavana.

Tutkielmassa esitetään, että funktionaalisen ohjelmoinnin hyödyt ovat todennäköisesti ainakin osittain seurausta paradigman tavasta käsitellä sivuvaikutuksia. Yhteyttä ei kuitenkaan voida käytettävissä olevan aineiston perusteella täysin osoittaa. Kuitenkin paradigman kiistattomien hyötyjen ansiosta funktionaalisen ohjelmoinnin todetaan olevan varteenotettava vaihtoehto imperatiiviselle paradigmalle.

Avainsanat: funktionaalinen ohjelmointi, funktionaaliset ohjelmointikielet, ohjelmointiparadigmat

Sisällysluettelo

1 Johdanto

Koko tietokoneiden historian ajan ohjelmointia ja sovelluskehitystä on pääasiassa lähestytty imperatiivisen ohjelmointiparadigman näkökulmasta. Aivan tietokoneiden kehityksen alkuajoista lähtien imperatiivisen paradigman rinnalla on kuitenkin kehittynyt funktionaalinen ohjelmointiparadigma, joka tarjoaa vaihtoehtoisen ja hyvin erilaisen lähestymistavan ohjelmointiin.

Funktionaalisella ohjelmoinnilla on pitkät perinteet yliopistoissa tutkimuskäytössä, ja vuosien varrella sitä on jonkin verran käytetty myös teollisuudessa. Suurimmaksi osaksi funktionaalinen ohjelmointi on kuitenkin jäänyt sovelluskehityksessä imperatiivisen ja oliopohjaisen paradigman varjoon. (Hinsen, 2009) Funktionaalisen ohjelmoinnin suosio on kuitenkin 2010-luvulla ollut selvästi kasvussa, minkä voi havaita esimerkiksi sen kasvaneesta edustuksesta sekä teollisuuden että tiedeyhteisön konferensseissa. Myös se, että monia funktionaalisten kielten ominaisuuksia on tuotu osaksi perinteisesti imperatiivisia ohjelmointikieliä, kertoo kasvaneesta kiinnostuksesta funktionaaliseen ajattelumalliin ja sen potentiaaliin. (Hu et al., 2015) Modernit tietokoneohjelmat voivat olla todella monimutkaisia kokonaisuuksia, ja muun muassa tämän monimutkaisuuden hallintaan on alettu etsiä työkaluja funktionaalisesta ohjelmoinnista (Benton & Radziwill, 2016).

Funktionaalista ohjelmointia on käsitelty paljon varsinkin akateemisessa kirjallisuudessa. Usein kuitenkin se, mitä funktionaalisella ohjelmoinnilla tarkalleen ottaen tarkoitetaan, määritellään valitettavan epäselvästi. Toisinaan taas funktionaalisen ohjelmoinnin esittely pelkistyy vain anekdooteiksi muuttumattomasta datasta ja sivuvaikutuksettomuudesta, jotka varsinkin imperatiiviseen ohjelmointiparadigmaan tottuneelle voivat kuulostaa täysin käsittämättömiltä. Pohjimmiltaan suurin haaste funktionaalisen ohjelmoinnin käsittelyssä lienee siinä, että funktionaalista ohjelmointia täytyy ymmärtää melko syvällisellä tasolla, jotta sen ominaisuudet alkaa nähdä rajoitteiden sijaan apuvälineinä monimutkaisten kokonaisuuksien hallinnassa. Funktionaalisen ohjelmoinnin opettelu on suuri ajankäytöllinen uhraus, jota on vaikea perustella, jos ei ole vakuuttunut sen hyödyistä.

Tämän tutkielman tutkimuskysymys on ”*Mitä funktionaalinen ohjelmointi on, ja mitä hyötyjä sen avulla voidaan saavuttaa?*” Funktionaalinen ohjelmointi on laaja ja monimutkainen aihepiiri, ja tämäkin tutkielma tarjoaa siihen vain pintaraapaisun. Paradigman syvällisen pohdinnan sijaan tässä tutkielmassa on pyritty löytämään keskeinen ydin funktionaalisen ohjelmoinnin perusideasta ja lähestymään konkreettisten hyötyjen kautta niitä syitä, joiden takia sen opiskelua ja käyttöä kannattaa harkita. Yhtenä tutkielman taustalla vaikuttavana tavoitteena on tarjota aihetta vähemmän tuntevalle väylä funktionaalisen ohjelmoinnin ymmärtämiseen.

Tutkielman luvussa 2 kuvataan menetelmä, jolla tutkimusaineisto on valittu. Keskeinen asiasisältö alkaa luvun 3 funktionaalisen ohjelmoinnin perusidean ja tyypillisten ominaisuuksien kuvauksella, ja etenee luvun 4 tieteellisissä artikkeleissa esiin nousseiden funktionaalisen ohjelmoinnin hyötyjen läpikäyntiin. Luvussa 5 pohditaan yhteyksiä havaittujen hyötyjen ja funktionaalisen ohjelmoinnin ominaisuuksien välillä, ja lopuksi luvussa 6 on tutkielman yhteenveto.

2 Tutkimusmenetelmä

Tämä tutkielma on muodoltaan kirjallisuuskatsaus. Katsauksen aineisto on ensisijaisesti koottu hakemalla aiheeseen liittyvää tieteellistä kirjallisuutta tietojenkäsittelyä käsitteleviä artikkeleita sisältävistä tietokannoista. Käytettyjä tietokantoja olivat ACM Digital Library, ProQuest – Computer Science Database, IEEE Electronic Library, ScienceDirect ja SpringerLink. Näiden tietokantojen lisäksi tuloksia on täydennetty hakemalla aineistoa myös Tampereen yliopiston Andor-hakupalvelusta sekä Google Scholarista.

Keskeisin käytetty hakutermi oli ”functional programming”, jota tarkennettiin lisäämällä AND-operaattorilla täydentäviä termejä kuten ”software development” ja ”benefits OR advantage”. Edellä kuvatuilla hauilla löytyneellä aineistolla kartoitettiin tutkielman keskeinen sisältö, minkä jälkeen joistakin esiin nousseista teemoista tehtiin lisäksi tarkempia hakuja. Tällaisia hakuja toteutettiin muun muassa termeillä ”’functional programming’ AND testability” ja ”’functional programming’ AND (parallelism OR parallel OR concurrency OR concurrent)”. Aineiston haut toteutettiin tammi- ja helmikuussa 2021.

Tutkielman aineistoksi päätyneiden artikkeleiden valinnassa on pyritty painottamaan uudempia kirjoituksia etenkin, jos ne käsittelivät aihetta käytännönläheisemmästä näkökulmasta. Teoriapainotteisten näkökulmien osalta julkaisuajankohdan merkitys tulkittiin vähäisemmäksi, joten aineistoon on päätynyt myös vanhempia artikkeleita. Hauissa nousi esille myös sellaista aineistoa, jossa joitakin tässä tutkielmassa käsiteltyjä teemoja pidettiin itsestäänselvyyksinä ilman merkittävää analyyttista tarkastelua. Tämänkaltaiset artikkelit eivät usein kuitenkaan olleet keskeiseltä sisällöltään tämän tutkielman kannalta mielenkiintoisia, joten ne eivät päätyneet tutkielmaan aineistoon.

Tutkielmassa on etsitty aineistoon valikoituneissa kirjoituksissa usein esiin nousevia tai muuten keskeisiä teemoja ja näkökulmia. Niiden perusteella on pyritty luomaan johdonmukainen kokonaisuus, joka tarjoaa tieteeseen pohjaavan katsauksen funktionaalisen ohjelmoinnin periaatteisiin ja sen käytön motiiveihin.

3 Funktionaalinen ohjelmointi

3.1 Funktionaalisen ohjelmoinnin perusajatus

Perinteisesti ohjelmointitermiä käytettäessä tarkoitetaan imperatiivista ohjelmointia. Imperatiivisen ohjelmoinnin keskeisenä ajatuksena on kirjoittaa tietokoneelle suoritettavaksi komentoja, jotka muokkaavat tietokoneen muistissa olevaa dataa, kunnes muistiin tallennettu data on halutunlaista. Imperatiivinen ohjelmointiparadigma on intuitiivinen ja se muistuttaa paljolti sitä, miten tietokone toimii laitetasolla. Imperatiivinen ohjelmointi on kuitenkin vain yksi ohjelmointiparadigma, ja sen rinnalla kehittynyt funktionaalinen ohjelmointi on hyvin erilainen tapa kehittää ohjelmia. (Hinsen, 2009) Funktionaalisessa ohjelmoinnissa kantavana ajatuksena on keskittyä siihen, *mitä* ohjelman halutaan tekevän sen sijaan, että lähdettäisiin liikkeelle siitä, *miten* ohjelma toteutetaan (Benton & Radziwill, 2016).

Funktionaalinen ohjelmointi perustuu pohjimmiltaan Alonzo Churchin 1930-luvulla kehittämään *lambdakalkyyliin* (lambda calculus / λ-calculus). Ensimmäisenä varsinaisena funktionaalisena ohjelmointikielenä pidetään John McCarthyn 1950-luvulla kehittämää Lisp-kieltä. Nimi Lisp on lyhenne sanoista *List Prosessing*. (Hinsen, 2009) Ensimmäinen Lispin versio ei sisältänyt lainkaan kääntäjää, vaan kieltä käännettiin konekielelle käsin. Myöhemmin McCarthyn oppilas Steve Russell havaitsi, että kielen rakenteen ansiosta sille saatiin ohjelmoitua tulkki pelkästään `eval`-funktion toteutuksella. Voidakseen käyttää funktioita toisten funktioiden parametreina McCarthy lainasi Churchin lambdakalkyylista perussyntaksin Lispin funktiolle. (McCarthy, 1978)

Nimensä mukaisesti funktionaalisen ohjelmoinnin keskeinen elementti on funktio. Funktionaalinen ohjelma rakentuu määrittelemällä `main`-funktio toisilla funktioilla ja niiden yhdistelmillä, jotka taas osaltaan määritellään toisilla funktioilla ja niiden yhdistelmillä. (Hughes, 1989) Funktionaalisessa ohjelmoinnissa funktio on hyvin samankaltainen käsite kuin diskreetissä matematiikassa (VanDrunen, 2017). Funktio saa parametreja ja palauttaa aina jonkin arvon, ja sen palauttamaan arvoon vaikuttavat ainoastaan sen saamat parametrit. Funktion suorittaminen ei myöskään saa muuttaa mitään tilaa tai arvoa tietokoneen muistissa. (Hinsen, 2009)

Funktion matemaattisen luonteen seurauksensa funktiokutsu voidaan ohjelmakoodissa aina korvata sen palautusarvolla. Tätä ominaisuutta kutsutaan funktion *puhtaudeksi* (purity) tai *viittausten läpinäkyvyydeksi* (referential transparency). Funktio ei siis tarkalleen ottaen saa parametreja ja palauta arvoja, vaan funktiokutsu tietyillä arvoilla on täysin ekvivalentti palautusarvonsa kanssa. (Hu et al., 2015)

Funktionaalisen ohjelmointiparadigman yhteys diskreettiin matematiikkaan tulee ilmi myös siinä, miten muuttujia käsitellään (VanDrunen, 2017). Funktionaalisessa ohjelmoinnissa ei ole lainkaan muuttujia siinä mielessä kuin ne imperatiivisessa ohjelmoinnissa käsitetään, koska niiden arvoa ei funktioiden puhtauden takia voisi koskaan muuttaa

(Hinsen, 2009). Funktionaalisessa ohjelmoinnissa myös data ja tietorakenteet ovat *muuttumattomia* (immutable). Imperatiivisessa ohjelmoinnissa on yleistä välittää funktioille viittauksia muistissa olevaan dataan, jonka tilaa funktio muokkaa. Tämä ei kuitenkaan lähtökohtaisesti ole funktionaalisessa ohjelmoinnissa sallittua, vaan funktion muokkaama data on aina muokattu kopio alkuperäisestä datasta.

3.2 Funktionaalisten ohjelmointikielten tyypillisiä ominaisuuksia

3.2.1 Sivuvaikutuksien käsittely

Funktionaalisen ohjelmointitavan yksi keskeisimpiä piirteitä se, miten siinä suhtaudutaan sivuvaikutuksiin. Funktion sivuvaikutuksella tarkoitetaan sitä, että funktio on paluuarvoksi evaluoitumisen lisäksi jonkinlaisessa vuorovaikutuksessa ympäristönsä kanssa. Tällaisia vuorovaikutuksia on esimerkiksi tilan muuttaminen tietokoneen muistissa, tekstin tulostaminen näytölle, syötteen lukeminen ja tiedostojen tai tietokantojen käsittely. Lähtökohtaisesti funktionaalinen ohjelma ei aiheuta lainkaan sivuvaikutuksia. Sivuvaikutuksettomat funktiot eivät varsinaisesti *tee* mitään, vaan ainoastaan evaluoituvat parametreista johdetuksi palautusarvoksi. Täysin sivuvaikutukseton ohjelma ei kuitenkaan voisi olla millään tavalla vuorovaikutuksessa muun maailman kanssa, minkä seurauksena sivuvaikutukseton ohjelma olisi käytännössä hyödytön. Funktionaalisissa kielissä sivuvaikutuksia pyritään välttämään, ja silloin kun ne ovat välttämättömiä, ne toteutetaan harkitusti vain tietyissä ohjelman osissa (Hinsen, 2009).

3.2.2 Anonyymit ja korkeamman asteen funktiot

Funktionaalissa ohjelmoinnissa on hyvin tyypillistä, että funktiot saavat parametrina toisia funktioita tai palauttavat niitä. Tällaisia funktioita kutsutaan *korkeamman asteen funktioiksi* (higher-order functions) (Hughes, 1989). Yleisesti käytetty `map`-funktio on korkeamman asteen funktio, joka saa parametrinaan toisen funktion sekä listan, jonka kaikille alkioille parametroitu funktio suoritetaan. Alla olevassa esimerkissä on esitetty Haskell-kielellä koodi, jossa ensin määritellään funktio `square`, joka korottaa saamansa parametrin toiseen potenssiin. Tämän jälkeen `square`-funktio suoritetaan listan `[1,2,3]` kaikille alkioille. Tuloksena saadaan uusi lista, joka sisältää kunkin alkuperäisen listan arvon korotettuna toiseen potenssin.

```
Prelude> let square x = x * x
Prelude> map square [1,2,3]
[1,4,9]
```

Korkeamman asteen funktioiden yhteydessä käytetään usein *anonyymejä funktioita* (anonymous functions / lambda expressions). Tällä tarkoitetaan sitä, että uusi funktio voidaan

esitellä nimettömänä keskellä muuta koodia. Edellinen esimerkki voidaan kirjoittaa korvaamalla `square`-funktio anonyymillä funktiolla seuraavalla tavalla.

```
Prelude> map (\x -> x * x) [1,2,3]
[1,4,9]
```

Korkeamman asteen funktioiden ja anonyymien funktioiden avulla voidaan monissa tapauksissa toteuttaa hyvin ilmaisuvoimaista koodia, ja tästä syystä vastaava toiminnallisuus on tuotu osaksi myös useimpia lähtökohtaisesti ei-funktionaalisia ohjelmointikieliä kuten Java, C++, C#, Python ja JavaScript (Hu et al., 2015).

3.2.3 Osittainen suorittaminen

Funktioita voidaan luoda myös *osittaisella suorittamisella* (partial application). Tällä tarkoitetaan sitä, että funktiota ei kutsuta sen kaikilla parametreilla, vaan vähintään yksi parametreista jätetään avoimeksi. Palautusarvona saadaan uusi funktio, joka ottaa vastaan jäljelle jääneet parametrit. Esimerkiksi `inc`-funktio, joka kasvattaa saamaansa kokonaislukuparametria yhdellä, voidaan toteuttaa Haskell-kielellä seuraavalla tavalla.

```
let inc = (+) 1
```

Tässä `inc` on osittain suoritettu +-operaatio, joka on saanut ensimmäiseksi parametrikseen numeron 1 ja odottaa toistaa toista parametria suorittaakseen laskutoimituksen loppuun. (Laufer & Thiruvathukal, 2009)

3.2.4 Funktioiden yhdistäminen

Funktionaalisessa ohjelmoinnissa uusia funktioita luodaan *yhdistetyillä funktioilla* (function composition). Funktioiden yhdistäminen funktionaalisissa ohjelmointikielissä toimii samalla tavalla kuin yhdistetty funktio diskreetissä matematiikassa. (VanDrunen, 2017)

Matematiikassa määritelmä yhdistetylle funktiolle esitetään seuraavalla tavalla.

$$(g \circ f)(x) = g\big(f(x)\big)$$

Tämä tarkoittaa, että $g \circ f$ on uusi funktio, joka on ekvivalentti sen kanssa, että ensin suoritetaan funktio f arvolle x, ja sen jälkeen funktio g arvolle, jonka f tuottaa. (Merikoski et al., 2004)

Funktioiden yhdistäminen on yksi funktionaalisen ohjelmoinnin keskeisiä tapoja käyttää uudelleen ja tuottaa ohjelmakoodia (Hughes, 1989). Funktioiden yhdistämistä voidaan käyttää esimerkiksi seuraavalla tavalla.

```
Prelude> map (inc . square) [1,2,3]
[2,5,10]
```

Haskell-kielessä ∘-symbolin vastine on normaali piste (.). Tässä esimerkissä on luotu uusi funktio yhdistämällä aiemmin esitellyt `inc`- ja `square`-funktiot. Yhdistämisen tuloksena saadaan funktio, joka laskee parametrinsa neliön ja kasvattaa neliöarvoa yhdellä. Kun yhdistetty funktio annetaan `map`-funktiolle parametriksi, se suoritetaan listan `[1,2,3]` kaikille alkioille. On tärkeää huomata, että funktioiden yhdistämisessä funktioiden järjestyksellä on väliä. Jos `square`- ja `inc`-funktiot olisi yhdistetty käänteisessä järjestyksessä, olisi listan alkioiden arvoa ensin kasvatettu yhdellä ja sen jälkeen korotettu toiseen potenssin, mistä olisi seurannut täysin eri lopputulos.

3.2.5 Laiska evaluointi

Funktioiden puhtaudesta ja viittausten läpinäkyvyydestä seuraa, että funktion suoritusajankohdalla ei ole merkitystä, jolloin funktiota ei tarvitse suorittaa ennen kuin sen palautusarvoa oikeasti tarvitaan. Tätä kutsutaan *laiskaksi evaluoinniksi* (lazy evaluation). Laiska evaluointi mahdollistaa muun muassa äärettömän pitkien listojen käyttämisen, sillä listaa evaluoidaan vain siihen asti kuin tarvitaan. (Laufer & Thiruvathukal, 2009)

Haskell-kielellä laiskaa evaluointia voidaan hyödyntää esimerkiksi `zip`-funktion kanssa seuraavalla tavalla.

```
Prelude> zip [1..] ["A", "B", "C", "D"]
[(1,"A"),(2,"B"),(3,"C"),(4,"D")]
```

Esimerkissä yhdistetään `zip`-funktiolla kaksi listaa, joista ensimmäinen sisältää numerot yhdestä äärettömään ja toinen aakkosten neljä ensimmäistä kirjainta. Koska funktio lopettaa kummankin listan *kulkemisen* (traversal) silloin, kun lyhyempi lista on käsitelty loppuun, evaluoidaan äärettömästä listasta vain neljä ensimmäistä alkiota.

3.2.6 Hindley–Milner -tyyppipäättely

Hindley–Milner -tyyppipäättely on tekniikka, jolla joissakin funktionaalisissa kielissä voidaan saavuttaa staattisen tyyppijärjestelmän edut menettämättä kuitenkaan dynaamisen tyyppijärjestelmän joustavuutta. Ohjelmointikielet monesti jaetaan staattisesti ja dynaamisesti tyypitettyihin kieliin. Dynaamisen tyyppijärjestelmän etuna katsotaan usein olevan kehittämisen nopeus ja joustavuus. Staattisen tyyppijärjestelmän eduiksi sen sijaan katsotaan ohjelman parempi toimintavarmuus, koska monet virheet voidaan löytää jo ohjelman kääntövaiheessa. Joissakin funktionaalisissa kielissä yhdistyy sekä dynaamisen että staattisen tyyppijärjestelmän hyödyt. (Laufer & Thiruvathukal, 2009) Hindley–Milner -tyyppipäättelyn avulla voidaan ohjelmakoodista jättää lähes kaikki tyypit merkitsemättä, mutta tyypit voidaan silti päätellä ohjelmakoodin toteutuksesta (Hinsen, 2009).

Hindley–Milner -tyyppijärjestelmä mahdollistaa myös geneeristen tyyppien käyttämisen. Esimerkiksi tietorakenne voidaan määritellä olemaan kokoelma tiettyä tyyppiä, jolla käytännössä tarkoitetaan, että kokoelman kaikki alkiot voivat olla mitä tahansa samaa tyyppiä. (Hu et al., 2015)

3.3 Mikä tekee ohjelmointikielestä funktionaalisen?

Funktionaalisten kielten välillä on suuriakin eroja, eivätkä kaikki funktionaalisiksi luokitellut kielet välttämättä toteuta kaikkia edellä kuvattuja funktionaaliselle paradigmalle tyypillisiä ominaisuuksia. Funktionaalisen ohjelmoinnin periaatteiden noudattamiseen ei välttämättä edes tarvita funktionaalista ohjelmointikieltä. Useita funktionaalisten kielten ominaisuuksia, kuten korkeamman asteen funktiot, esiintyykin myös monissa ei-funktionaalisissa kielissä. Funktionaalisia kieliä kuitenkin yhdistää se, että ne kannustavat ja joissain tapauksissa myös pakottavat käyttämään funktionaalista ohjelmointi- ja ajattelutapaa (Hu et al., 2015).

Funktionaalisten ohjelmointikielten merkittävimpiä yhdistäviä tekijöitä on niiden suhtautuminen sivuvaikutuksiin. Vaikka kielten välillä on eroja siinä, miten sivuvaikutuksia toteutetaan, on oleellista, että sivuvaikutukset toteutetaan hallitusti ohjelman tietyissä osissa. Tällöin muu osa ohjelmasta pysyy sivuvaikutuksista puhtaana. Suurin osa funktionaalisista ohjelmointikielistä jättää vastuun sivuvaikutusten hallinnasta ohjelmoijalle, mutta puhtaat kielet kuten Haskell mahdollistavat sivuvaikutukset vain siihen tarkoitetun erillisen rakenteen avulla. (Hinsen, 2009)

Funktionaaliset kielet voidaan karkeasti jakaa kolmeen ryhmään. Ensimmäisen ryhmän muodostavat Lisp-johdannaiset kielet. Lispissä ja sen eri murteissa kantavana ajatuksena on, että myös koodi on dataa ja Lispin syntaksi rakentuukin samalla tavalla monitasoisista listoista kuin tietorakenteet, joita sillä käsitellään. Lispissä tietotyypit ovat dynaamisesti tyypitettyjä samoin kuten esimerkiksi JavaScriptissä ja Pythonissa. Nykyisin käytetyimpiä Lispin murteita ovat Common Lisp, Scheme ja Clojure. (Hinsen, 2009)

Toisen ryhmän muodostavat ML:stä johdetut kielet. ML kehitettiin alun perin 1970-luvulla Edinburghin yliopistossa ja nykypäivänä sen merkittävimpiä johdannaisia ovat Standard ML sekä OCaml. Lispistä ja sen murteista poiketen ML ja siitä johdetut kielet ovat staattisesti tyypitettyjä ja niissä sovelletaan Hindley–Milner -tyyppipäättelyä. (Hinsen, 2009) Myös Microsoftin kehittämän .NET-alustalla toimivan funktionaalisen ohjelmointikielen F#:n voidaan katsoa kuuluvan ML-perheeseen, vaikka se on myös saanut vaikutteita muun muassa Haskellista. (McCaffrey & Bonar, 2010)

Kolmannen ryhmän muodostavat puhtaat funktionaaliset kielet kuten Haskell. Haskell kehitettiin alun perin yliopistomaailmassa yhteiseksi standardiksi helpottamaan tutkijoiden yhteistyötä. ML-kielten tavoin Haskell hyödyntää Hindley–Milner -tyyppipäättelyä. Haskellin merkittävimpiä erityispiirteitä on laiska evaluointi sekä täydellinen funktioiden puhtauden noudattaminen. (Hinsen, 2009)

4 Funktionaalisen ohjelmoinnin hyötyjä

4.1 Testattavuus

Nykyaikana erilaiset ohjelmistot ovat merkittävä osa jokapäiväistä elämää. Niiden avulla voidaan luoda hyvinvointia ja parantaa tuottavuutta, mutta väärin toimivat tai virheelliset ohjelmat tuottavat paljon päänvaivaa ja pahimmillaan jopa vaarantavat henkiä. Asianmukainen testaaminen on yksi tärkeimpiä vaiheita ohjelman toimivuuden takaamisessa.

Imperatiivisen ohjelman testaaminen on usein hankalaa. Ohjelma saattaa esimerkiksi vaatia taustalleen tietynlaisen tietokannan tai tiedostorakenteen, ja iso osa testausprosessista muodostuu sopivan ympäristön eli alkutilan luomisesta. Kun alkutila on luotu, voidaan ohjelma suorittaa, minkä jälkeen tarkastellaan ohjelman tilaan tekemiä muutoksia. Ohjelman testaaminen on työläs prosessi, ja ohjelman suorituksen jälkeisestä tilasta voi monesti olla hankala päätellä, mitä tilan muutoksia mikäkin osa ohjelmasta on tehnyt. (Hu et al., 2015)

Funktionaalinen ohjelmointi tarjoaa tehokkaita työkaluja ohjelmakoodin testaamiseen ja se soveltuu erityisen hyvin käytettäväksi *testivetoisen kehitysmallin* (test-driven development, TDD) kanssa (Benton & Radziwill, 2016). Testivetoisella kehityksellä tarkoitetaan sitä, että ohjelma jaetaan mahdollisimman pieniin itsenäisesti toteutettaviin yksiköihin, ja näiden yksiköiden toiminnallisuus määritellään toteuttamalla niille testiohjelmat ennen varsinaisen ohjelmakoodin kirjoitusta. Testit ohjaavat ohjelman kirjoittamista ja helpottavat kehittäjän työskentelyä, sillä koodin toimivuus voidaan aina nopeasti varmistaa testien avulla. Imperatiivisessa olio-ohjelmoinnissa itsenäisesti toteutettava yksikkö on usein luokka tai sen yksittäinen metodi. (Janzen & Saiedian, 2005) Funktionaalisessa ohjelmoinnissa funktio muodostaa luonnollisen itsenäisesti toteutettavan yksikön. Funktion testaamista varten ei tarvitse luoda ympäristöä, koska se ei voi aiheuttaa sivuvaikutuksia, ja kunkin funktion palautusarvo riippuu ainoastaan sen saamista parametreista. (Benton & Radziwill, 2016)

Haskell-kielelle on kehitetty paljon käytetty QuickCheck-kirjasto ohjelmakoodin testaamista varten. Kirjaston avulla voidaan testata funktioita määrittelemällä niiden toiminnallisuus sääntöpohjaisesti. Yksinkertaisena esimerkkinä voidaan tarkastella `reverse`-funktiota, jonka on tarkoitus kääntää merkkijonon merkit tai listan alkiot käänteiseen järjestykseen. Funktio voidaan määritellä seuraavilla säännöillä.

```
reverse [] = []
reverse [x] = [x]
reverse (xs ++ ys) = reverse ys ++ reverse xs
reverse (reverse xs) = xs
```

Ensimmäinen ja toinen sääntö tarkoittavat sitä, että jos funktion parametrina saama lista on tyhjä tai se sisältää vain yhden alkion, on tulos sama kuin alkuperäinen lista. Kolmas sääntö tarkoittaa, että käännetty lista voidaan tuottaa jakamalla alkuperäinen lista mistä tahansa kohdasta alku- ja loppuosiin ja liittämällä `reverse`-funktiolla itsellään käännetyt osat yhteen siten, että käännetty loppuosa tulee tuloksen alkuun ja alkuosa loppuun. Neljännellä säännöllä tarkoitetaan, että kaksi kertaa käänteiseen järjestykseen käännetyn listan tulee aina olla sama kuin alkuperäinen, kääntämätön lista. Kolme ensimmäistä sääntöä riittävät määrittelemään täydellisesti `reverse`-funktion toiminnallisuuden. Neljännellä säännöllä saadaan varmistettua, että funktion toteutus toimii oikein. QuickCheck-kirjasto testaa funktiota annettua määritelmää vasten erilaisilla satunnaisesti generoiduilla syötteillä ja ilmoittaa, jos jollain syötteellä annettu määritelmä ei toteudu. Virheen aiheuttanut syöte voidaan tutkia yksittäistapauksena, ja funktion toteutus voidaan korjata suhteellisen vaivattomasti. (Hu et al., 2015)

Automaattitestaus suurella määrällä satunnaissyötteitä on hyvin tehokasta, ja siten voidaan monesti löytää sellaisia virheitä, joita ihmistestaaja ei tulisi koskaan ajatelleeksi. QuickCheckin kanssa samaan periaatteeseen pohjautuvia testauskirjastoja on luotu useimmille funktionaalisille ohjelmointikielille kuten F#:lle, Clojurelle ja Scalalle, mutta myös ei-funktionaalisille kielille kuten Javalle ja Golle. (Hu et al., 2015)

4.2 Ohjelmakoodin ja -logiikan oikeellisuus

Ohjelmakoodin testaaminen on äärimmäisen tärkeää toimivien ohjelmien rakentamisessa, mutta testaamiseen liittyy perustavanlaatuinen ongelma. Testaamisella ei ikinä voida varmistua ohjelmakoodin virheettömyydestä, vaan testaamisen seurauksena voidaan ainoastaan tietää, että ohjelma toimii oikein testeissä esiin tulleissa tilanteissa. (Hu et al., 2015) Tässäkin asiassa funktionaalisen ohjelmoinnin läheinen yhteys matematiikkaan tuottaa ratkaisuja. Funktionaalisen ohjelman rakenne mahdollistaa imperatiivista ohjelmaa paremman lähtökohdan ohjelmakoodin oikeellisuuden vahvistamiseen (Page, 2001). Funktionaaliset ohjelmat on usein määritelty rekursiivisesti, minkä ansiosta niiden oikeellisuus voidaan todistaa rakenteellisen induktion avulla. Induktiotodistusta käytetään usein varmistamaan kriittisten ohjelmistojärjestelmien oikeellisuus. (Hu et al., 2015)

Ohjelmakoodin sekä sen logiikan oikeellisuuden osoittaminen nousee useissa lähteissä esille funktionaalisen ohjelmoinnin keskeisenä hyötynä verrattuna imperatiiviseen ohjelmointiin. Hallitsemattomat sivuvaikutukset tekevät imperatiivisen koodin toiminnasta arvaamatonta. Funktionaalisessa ohjelmoinnissa sen sijaan viittausten läpinäkyvyys mahdollistaa koodin toiminnan yksiselitteisen loogisen tarkastelun (Laufer & Thiruvathukal, 2009).

4.3 Modulaarisuus

Modulaarisuudella tarkoitetaan sitä, että ohjelmakoodi jaetaan pieniin itsenäisiin osiin eli moduuleihin. Näin pyritään pitämään koodi helposti ylläpidettävänä, ja toisaalta modulaarisuudella tavoitellaan myös koodin laajempaa uudelleenkäyttöä. Modulaarisuus on ollut keskeinen tavoite koko ohjelmointiparadigmojen ja -kielien historian ajan. (Figueroa & Robbes, 2015)

John Hughes (1989) esittää kirjoituksessaan *Why Functional Programming Matters*, että korkeamman asteen funktioilla ja laiskalla evaluoinnilla on merkittävä rooli modulaarisen koodin kehittämisessä. Figueroa ja Robbes (2015) kuitenkin arvioivat, että funktionaalisen ohjelmoinnin hyötyjä modulaarisuuden suhteen saatetaan liioitella, ja heidän tekemänsä koodianalyysi *GHC Haskell compiler* -projektista viittaisi siihen, että funktionaalinen ohjelmointi ei ainakaan automaattisesti johda parempaan modulaarisuuteen.

Funktionaalinen ohjelmointi kuitenkin tarjoaa koodin uudelleenkäyttöön hyödyllisiä mekanismeja. Aikaisemmin esitelty `map`-funktio sekä `reduce` toimivat tästä hyvinä esimerkkeinä, sillä ne piilottavat taakseen koko tietorakenteen kulkemisen, ja ohjelmoijan työksi jää vain parametroida funktiona varsinainen kussakin tietorakenteen alkiossa suoritettava toiminto. Vastaava toiminnallisuus monesti ohjelmoidaan imperatiivisessa ohjelmoinnissa `for`- tai `while`-silmukalla. Tällä tavoin funktionaaliset ohjelmointimallit mahdollistavat korkeamman abstratraktiotason toiminnallisuuden uudelleenkäyttämisen (Hu et al., 2015).

4.4 Ohjelmakoodin ilmaisuvoima

Funktionaalista ohjelmointia käsiteltäessä monesti nousee esille funktionaalisten kielten *ilmaisuvoima*. Yleensä kielen ilmaisuvoima käsitteenä yksinkertaistuu siihen, kuinka paljon koodirivejä tarvitaan jonkin ominaisuuden toteuttamiseen. Varsinaista tutkimusta aiheesta ei juuri ole tehty, ja maininnat ilmaisuvoimasta ovat yleensä vain sivuhuomioita funktionaalisten kielten käyttöä käsiteltäessä. Aihe on kuitenkin merkittävä, koska ilmaisuvoimalla voidaan ajatella olevan suora vaikutus ohjelmakoodin luettavuuteen.

Gat (2000) vertailee artikkelissaan Common Lisp- ja Scheme-kielillä kirjoitettuja ohjelmia vastaaviin Java-, C- ja C++-toteutuksiin ja toteaa, että Lispillä kirjoitettujen ohjelmien pituudet vaihtelivat 51 rivistä 182 riviin. Vastaavien ohjelmien pituudet Javalla, C:llä ja C++:lla toteutettuna vaihtelivat 107 rivistä 614 riviin. Samansuuntaisiin tuloksiin on päätynyt myös Mäki (2019) diplomityönään toteuttamassa projektissa, jossa alun perin Javalla toteutetun ohjelman osakokonaisuus uudistettiin käyttämällä pääasiallisena ohjelmointikielenä Clojurea. Uudistuksen jälkeen järjestelmän koodirivien määrä oli vain noin viidesosa alkuperäisen järjestelmän rivimäärästä. Yhdeksi syyksi uudistetun järjestelmän pienempään koodimäärään Mäki tulkitsee ohjelmointikielen vaihtamisen funktionaaliseen.

4.5 Tuottavuus

Monessa yhteydessä nousee esiin väite, että funktionaalisen kielen käyttö on tuottavampaa kuin imperatiivisen kielen. Tuottavuudella tarkoitetaan ohjelman kehittämiseen kuluvaa aikaa. Luvussa 4.4 käsitellyn koodin ilmaisuvoiman lisäksi Gatin (2000) tutkimuksen mukaan myös ohjelmien toteuttamiseen kulunut aika oli Lisp-kielillä huomattavasti lyhyempi, ja siinä oli vähemmän vaihtelua kuin imperatiivisilla kielillä. Lispillä ohjelmien toteuttamiseen kului 2–8,5 tuntia vastaavien toteutusaikojen ollessa C:llä ja C++:lla 3–25 ja Javalla 4–63 tuntia. Gat myös korostaa, että Lispin parempaa tuottavuutta ei voi selittää kehittäjien kokemuksella, sillä Lispillä ohjelmat toteuttaneet kehittäjät olivat Java, C- ja C++-kehittäjiä kokemattomampia.

Funktionaalisen ohjelmointikielen käytön tuottavuuden nostavat esille myös Scott ja muut (2010) tapaustutkimuksessaan, jossa selvitettiin funktionaalisen ohjelmointikielen käytön vaikutuksia suuressa tuotekehitysprojektissa. Tutkimuksen mukaan OCaml-kielen käyttö paransi merkittävästi kehitystiimin tuottavuutta ja tehokkuutta ohjelmoinnissa verrattuna siihen, että ohjelmointikielenä olisi käytetty vaatimukset täyttävää imperatiivista kieltä kuten C++:aa tai Pythonia.

4.6 Rinnakkaislaskenta

Funktionaalisen ohjelmoinnin konkreettisimpia hyötyjä on sen tarjoamat mahdollisuudet rinnakkaislaskennassa. Englanninkielisessä kirjallisuudessa käytetään termejä *concurrency* ja *parallelism*, jotka molemmat suomentuvat rinnakkaisuudeksi mutta tarkoittavat eri asioita. Concurrency tarkoittaa sitä, että ohjelma käsittelee samaa dataa useissa rinnakkaisissa säikeissä, kun taas parallelism-termillä tarkoitetaan suoritettavan tehtävän jakamista useaan rinnakkain ajettavaan aliprosessiin (Hinsen, 2009). Tässä tutkielmassa viitataan parallellism-termiin rinnakkaislaskennalla ja concurrency-termiin monisäikeisyydellä.

Tietokoneiden prosessoreiden laskentateho ei kasva enää samalla tavalla kuin aikaisemmin, minkä johdosta tehon lisäys tapahtuu nykyään lähinnä kasvattamalle prosessoreiden ytimien määrää. Jotta prosessoriytimien lisäämisellä saavutettu laskentateho saadaan ohjelmissa hyödynnettyä, on koodissa hyödynnettävä rinnakkaislaskentaa ja monisäikeisyyttä. Näitä ohjelmointimalleja kuitenkin pidetään yleisesti erittäin haastavina. (Hinsen, 2009)

Funktionaalinen ohjelmointi tarjoaa hyvät puitteet varsinkin rinnakkaislaskennan toteuttamiseen. Funktioiden puhtaudesta seuraa, että funktioiden suoritusajankohdalla tai -järjestyksellä ei ole vaikutusta niiden palauttamiin arvoihin. Näin ollen ohjelmasta on helposti tunnistettavissa osat, jotka voidaan suorittaa rinnakkain, koska voidaan varmasti tietää, ettei prosessien tulokseen vaikuta mikään muu kuin niille annetut parametrit. Imperatiivisessa ohjelmoinnissa vastaava jako rinnakkain suoritettaviin prosesseihin vaatisi

mahdollisesti hyvin monimutkaisen analyysin siitä, etteivät rinnakkain laskettavat ohjelman osat vaikuta toisiinsa. (Hu et al., 2015)

Konkreettisena esimerkkinä rinnakkaislaskennan soveltamisessa Hu ja muut (2015) esittävät *pikalajittelualgoritmin* (quicksort). Pikalajittelun ensimmäisen iteraation tuloksena on kaksi joukkoa, jotka lajitellaan erikseen samalla algoritmilla. Nämä joukot voidaan lajitella samanaikaisesti rinnakkain eri prosessoriytimillä, koska tiedetään, etteivät aliprosessit vaikuta toisiinsa millään tavalla. Niiden tulokset vain yhdistetään lopuksi. Vähänkään suurempien joukkojen lajittelussa voidaan helposti hyödyntää niin suurta määrää ytimiä, kuin on saatavilla. Tämän seurauksena tietokone suoriutuu lajitteluprosessista rinnakkaislaskentaa hyödyntämällä huomattavasti sarjalaskentaa nopeammin.

5 Keskustelu

Vaikka funktionaalista ohjelmointia on käsitelty tieteellisissä julkaisuissa melko laajasti, on näkökulma pääasiassa teoreettinen, eikä laajamittaista empiiristä tutkimusta sen vaikutuksista sovelluskehitykseen ei ole tehty. Teollisuudessa on alettu tunnistaa funktionaalisen ohjelmoinnin etuja, mutta niiden käsittely akateemisessa kirjallisuudessa pelkistyy monesti lähinnä anekdooteiksi menestystarinoista, ja paradigman kriittinen tarkastelu lähestulkoon loistaa poissaolollaan. Tästä syystä tämänkin tutkielman näkökulma on pääasiassa teoreettinen. Teoreettisesta lähtökohdasta huolimatta on funktionaalisen paradigman käytöstä löydettävissä kiistattomia käytännön hyötyjä.

Funktionaalisten kielten välillä on joissakin tapauksissa suuriakin eroja, joten funktionaalisen paradigman käsittely vain yhtenä kokonaisuutena ei välttämättä kaikissa tapauksissa tuota riittävän tarkkaa analyysia sen käytön eduista. Kielten eroista merkittävimpiä lienevät erilaiset käytännöt sivuvaikutusten käsittelyssä ja erot tyyppijärjestelmissä. Merkittävä osa käsitellystä kirjallisuudesta tarkastelee funktionaalista ohjelmointia jonkin tietyn kielen tai kieliryhmän näkökulmasta, joten tuloksia ei aina voida yksiselitteisesti yleistää koskemaan koko paradigmaa.

Tässä tutkielmassa on pyritty keskittymään funktionaalisten kielten erojen sijaan niiden yhtäläisyyksiin. Funktionaalisen ohjelmointitavan keskeisimpiä piirteitä on se, että sivuvaikutuksia pyritään välttämään. Silloinkin kun sivuvaikutukset ovat välttämättömiä, ne toteutetaan hallitusti pitäen mahdollisimman suuri osa ohjelmasta niistä puhtaana. Kaikkien tässä tutkielmassa esiin nousseiden hyötyjen voidaan ainakin jossain määrin tulkita olevan seurausta funktionaalisen ohjelmoinnin tavasta käsitellä sivuvaikutuksia. Testattavuus ja rinnakkaislaskenta ovat tästä ilmeisiä esimerkkejä, mutta myös muun muassa modulaarisuuden ja ohjelmakoodin uudelleenkäytön voi katsoa helpottuvan sillä, että sivuvaikutukset ovat tarkasti kontrolloituja. Sivuvaikutusten poissaolon ansiosta voidaan varmasti tietää, ettei käyttöön otettava koodi tee mitään odottamatonta. Myös Gatin (2000) esille nostamien funktionaalisen koodin ilmaisuvoiman ja sovellusten lyhyemmän

kehitysajan voi olettaa ainakin osaksi olevan seurausta tehokkaan modularisoinnin mahdollistavista malleista kuten tietorakenteiden kulkemisen abstrahoinnista. Tarkemman empiirisen tutkimuksen puuttuessa yhteys sivuvaikutusten välttämisen ja funktionaalisen ohjelmoinnin hyötyjen välillä ovat kuitenkin vain spekulointia.

Luvussa 3.3 nostettiin sivuhuomiona esille mielenkiintoinen yksityiskohta siitä, ettei funktionaalisen ohjelmointitavan soveltaminen välttämättä vaadi funktionaaliseksi luokiteltua kieltä. Tämä luonnollisesti herättää kysymyksen siitä, miksi ohjelmoijan kannattaisi rajata käytössä oleva tekniikkavalikoima vain funktionaalisen ohjelmoinnin tekniikoihin. Funktionaalinen ohjelmointi on kiistatta monissa tilanteissa rajoittavaa, ja John Hughes (1989) vertaakin humoristisesti funktionaalisen ohjelmointikielen käyttäjää keskiaikaiseen munkkiin, joka hyveellisyyttä tavoitellakseen kieltäytyy elämän nautinnoista. Vertaus on siinä mielessä osuva, että vaikka monia funktionaalisen ohjelmoinnin tekniikoita on tuotu myös osaksi imperatiivisia kieliä, funktionaalisen ohjelmoinnin todelliset hyödyt ovat saavutettavissa vasta silloin, kun ohjelman tai sen osan tiedetään varmasti noudattavan funktionaalista ohjelmointitapaa. Todellisessa elämässä ohjelmoinnin aikataulupaineiden seurauksena, jos ohjelmointikieli sen mahdollistaa, koodiin päätyy nopeita, mutta pitkällä aikavälillä kestämättömiä ratkaisuja. Funktionaalisen ohjelmointikielen valinnalla voidaan ohjata parempaan ohjelman suunnitteluun ja minimoida huonojen toteutusratkaisujen määrää, mikä voi pidentää sovellusten elinkaarta ja tuottaa pitkällä aikavälillä huomattavia kustannussäästöjä.

Funktionaalisen ohjelman suunnittelu ja toteutus poikkeaa lähtökohdiltaan imperatiivisen ohjelman kehittämisestä. Tässä tutkielmassa ainoastaan sivuttiin aihetta Bentonin ja Radziwillin (2016) esittämällä toteamuksella siitä, että funktionaalisessa ohjelmoinnissa ei lähdetä liikkeelle toteutuksen teknisistä yksityiskohdista, vaan siinä keskitytään määrittelemään haluttu toiminnallisuus. Funktionaalisen ohjelman suunnittelu- ja toteutusprosessin yksityiskohtaisempi tarkastelu jää tulevien tutkimusten tehtäväksi.

6 Yhteenveto

Kiinnostus funktionaaliseen ohjelmointiin kasvaa koko ajan, ja siitä haetaan apua moniin ohjelmistokehityksen haasteisiin. Paradigman pitkä historia tiedeyhteisön tutkimustyökaluna luo otollisen pohjan tarkastella, mitä funktionaalinen ohjelmointi tarkalleen ottaen on, ja mitä sen avulla voi saavuttaa. Tässä tutkielmassa on esitelty funktionaalisen ohjelmoinnin keskeisiä periaatteita ja tekniikoita, sekä kartoitettu sen käytöllä saavutettavia hyötyjä sovelluskehityksessä.

Funktionaalinen ohjelmointi on ohjelmointiparadigma, jonka kantavana ajatuksena on ohjelmien tuottaminen luomalla ja yhdistämällä funktioita. Imperatiivisen ohjelmoinnin funktiokäsitteestä poiketen funktionaalisen ohjelmoinnin funktio muistuttaa ominai-

suuksiltaan diskreetin matematiikan funktiota eli kuvausta lähtöjoukon ja tulosjoukon välillä. Funktion matemaattisen luonteen seurauksena sivuvaikutusten toteuttaminen on haaste funktionaalisessa paradigmassa. Funktionaalisten kielten välillä on eroja siinä, miten ne lähestyvät sivuvaikutusten luomaa haastetta, mutta yhteistä niille on se, että sivuvaikutukset pyritään toteuttaman hallitusti ja rajaamaan vain tiettyihin ohjelman osiin.

Tässä tutkielmassa esitettiin funktionaalisen ohjelmoinnin hyötyjen olevan pääasiassa seurausta sivuvaikutusten hallitusta toteuttamisesta. Kun sivuvaikutukset rajataan tarkasti, pysyy merkittävä osa ohjelmasta niistä puhtaana. Ohjelman sivuvaikutuksettomien osien toiminta on helposti ennakoitavaa, sillä ne toimivat samoilla syötearvoilla aina samalla tavalla. Tämä mahdollistaa esimerkiksi ohjelman paremman testattavuuden ja sen toiminnan oikeellisuuden loogisen tarkastelun. Sivuvaikutusten välttäminen myös jakaa ohjelman luonnollisiin toisistaan riippumattomiin osakokonaisuuksiin, mikä helpottaa niiden uudelleenkäyttöä ja toisaalta mahdollistaa turvallisemman rinnakkaislaskennan ja monisäikeisyyden toteuttamisen.

Funktionaalisen ohjelmointiparadigman käytöstä seuraa kiistatta monia hyötyjä. Se, onko vähemmän tunnetun ohjelmointikielen valinta ohjelmistoprojektiin sen hyödyistä huolimatta perusteltua, riippuu luonnollisesti muistakin tekijöistä kuin itse kielen tai paradigman ominaisuuksista. Funktionaalinen ohjelmointi tarjoaa kuitenkin ohjelmistokehitykseen hyvän vaihtoehtoisen lähestymistavan, jota ei kannata tekniikoiden valinnan yhteydessä sivuuttaa.

Lähdeluettelo

Benton, M. C., & Radziwill, N. M. (2016). *Improving Testability and Reuse by Transitioning to Functional Programming*. http://arxiv.org/abs/1606.06704

Figueroa, I., & Robbes, R. (2015). Is functional programming better for modularity? *PLATEAU 2015 - Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, 49–52. https://doi.org/10.1145/2846680.2846689

Gat, E. (2000). Point of view: Lisp as an alternative to Java. *Intelligence*, *11*(4), 21–24. https://doi.org/10.1145/355137.355142

Hinsen, K. (2009). The promises of functional programming. *Computing in Science and Engineering*, *11*(4), 86–90. https://doi.org/10.1109/MCSE.2009.129

Hu, Z., Hughes, J., & Wang, M. (2015). How functional programming mattered. *National Science Review*, *2*(3), 349–370. https://doi.org/10.1093/nsr/nwv042

Hughes, J. (1989). Why functional programming matters. *Computer Journal*, *32*(2), 98–107. https://doi.org/10.1093/comjnl/32.2.98

Janzen, D., & Saiedian, H. (2005). Test-driven development: concepts, taxonomy and future directions. *IEEE Computer*, *38*(9), 43–50. https://doi.org/10.1109/MC.2005.314

Laufer, K., & Thiruvathukal, G. K. (2009). Scientific programming: The promises of

typed, pure, and lazy functional programming: Part II. *Computing in Science and Engineering*, *11*(5), 68–75. https://doi.org/10.1109/MCSE.2009.147

Mäki, J. (2019). *Web-ohjelmiston uudistaminen funktionaalisella paradigmalla* [Diplomityö, Tampereen yliopisto]. http://urn.fi/URN:NBN:fi:tuni-201910234065

McCaffrey, J. D., & Bonar, A. (2010). A case study of the factors associated with using the F# programming language for software test automation. *ITNG2010 - 7th International Conference on Information Technology: New Generations*, 1009–1013. https://doi.org/10.1109/ITNG.2010.253

McCarthy, J. (1978). History of LISP. *SIGPLAN Notices*, *13*(8), 217–223. https://doi.org/10.1145/800025.808387

Merikoski, J., Virtanen, A., & Koivisto, P. (2004). *Johdatus diskreettiin matematiikkaan*. WSOY.

Page, R. (2001). Functional programming, and where you can put it. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, *36*(9), 19–24. https://doi.org/10.1145/609769.609771

Scott, D., Sharp, R., Gazagnaire, T., & Madhavapeddy, A. (2010). Using functional programming within an industrial product group: Perspectives and perceptions. *ACM SIGPLAN Notices*, *45*(9), 87–92. https://doi.org/10.1145/1932681.1863557

VanDrunen, T. (2017). Functional programming as a discrete mathematics topic. *ACM Inroads*, *8*(2), 51–58. https://doi.org/10.1145/3078325