



TAMPEREEN TEKNILLINEN YLIOPISTO

Arttu Kaipainen

Funktionaalinen ohjelmointi web-ohjelmistokehityksessä

Diplomityö

Tarkastaja: Prof. Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
8. kesäkuuta 2016

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

Arttu Kaipainen: Funktionaalinen ohjelmointi web-ohjelmistokehityksessä

Diplomityö, 50 sivua

Joulukuu 2016

Pääaine: Ohjelmistotuotanto

Tarkastajat: Prof. Kari Systä

Avainsanat: Funktionaalinen ohjelmointi, Ohjelmistotuotanto, Web-ohjelmointi

Valtaosa nykypäivänä luotavista ohjelmista toimii internetiin perustuen. Verkkosivut ovat ajan myötä kehittyneet staattisista HTML-sivuista kokonaisvaltaisiksi ohjelmiksi, jotka suoritetaan palvelimen sijaan käyttäjän verkkoselaimessa. Web-ohjelmointiin on muodostunut useita menetelmiä, joista reaktiivinen ohjelmointi on yksi suosituimmista.

Funktionaalisen ohjelmoinnin alkuperä on 1930-luvulla kehitetyssä lambdakalkyyllissä ja sitä ennen matematiikassa. Sen periaatteena on matemaattisen funktion käsite. Funktionaalisten ohjelmointikielien kehitys alkaa 1950-luvun Lispistä ja jatkuu edelleen nykypäivän Clojureen, Scalaan sekä Haskellin.

Tässä diplomityössä tutkitaan funktionaalisen ohjelmoinnin soveltuvuutta nykyaikaiseen web-ohjelmointiin. Tutkimusta varten on suoritettu sekä haastattelututkimus että kyselytutkimus Solita Oy:n työntekijöiden keskuudessa. Tutkimukseen on valittu työntekijöitä, joilla on kokemusta web-ohjelmoinnista sekä funktionaalisilla että imperatiivisilla kielillä.

Työn tuloksena todetaan, että funktionaalinen ohjelmointi soveltuu web-ohjelmistokehitykseen erittäin hyvin. Monet funktionaalisen ohjelmoinnin periaatteista ja menetelmistä sopivat luonnostaan web-ohjelmointiin, ja vaikutus ohjelmiston laatuun on muutenkin huomattava. Funktionaalisen ohjelmoinnin riskit ovat lähinnä tekijöiden löytämisessä ja kouluttamisessa.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Information Technology

Arttu Kaipainen: Functional programming in web software development

Master of Science Thesis, 50 pages

December 2016

Major: Software Engineering

Examiners: Prof. Kari Systä

Keywords: Functional programming, Software development, Web software

A majority of computer programs today are based on the internet. Web pages have evolved from static HTML pages into full-fledged programs, that are executed on the user's web browser instead of the web server. Multiple methods have been developed for web programming, and reactive programming is one of the most popular ones.

The origin of functional programming is in lambda calculus, developed in the 1930s, and in mathematics before that. Its basic principle is the concept of a mathematical function. The evolution of functional programming languages starts from Lisp in the 1950s, continuing until today's Clojure, Scala and Haskell.

In this thesis, the suitability of functional programming in modern web software development is studied. Both interviews and a questionnaire were performed in Solita Ltd. The participants had experience in web software development with both functional and imperative programming languages.

The conclusion of this thesis is that functional programming is very suitable for web software development. Many of the principles and methods of functional programming are naturally suited for web software development, and the effect on software quality is also noticeable. The risks of functional programming are mostly in finding and training the programmers.

ALKUSANAT

Tämä diplomityö on syntynyt Solita Oy:ssä sekä tekijän että yrityksen mielenkiinnon kohteena. Työssä tarkastellaan funktionaalisen ohjelmoinnin soveltuvuutta nykyaikaiseen web-ohjelmistokehitykseen.

Haluan kiittää työn ohjauksesta professori Kari Systää. Kiitokset kuuluvat myös Solita Oy:lle mahdollisuudesta työn tekemiseen sekä DI Antti Virtaselle ja DI Jouni Honkalalle arvokkaista näkemyksistä ja avusta. Erityiset kiitokset haluan osoittaa esimiehelleni, DI Ari Ruotsalaiselle jatkuvasta kannustamisesta ja tuesta diplomityön tekemisen aikana.

Tampereella 30.12.2016

Arttu Kaipainen

SISÄLLYS

1. Johdanto	1
2. Funktionaalinen ohjelmointi	3
2.1. Funktionaalisen ohjelmoinnin periaatteet	3
2.2. Funktionaalisen ohjelmoinnin historia	6
2.2.1. Lambdakalkyyli	6
2.2.2. Lisp	8
2.2.3. ML	10
2.3. Funktionaalinen ohjelmointi nykyään	11
2.3.1. Clojure ja ClojureScript	11
2.3.2. Scala	12
2.3.3. Java 8	13
2.3.4. F#	14
2.3.5. Haskell	14
3. Web-ohjelmointi	16
3.1. Web-palvelin	16
3.2. Web-käyttöliittymä	17
3.3. Funktionaalinen web-ohjelmointi	19
3.4. Funktionaalinen reaktiivinen ohjelmointi	20
4. Tutkimus	23
4.1. Tutkimusmenetelmä	23
4.1.1. Haastattelututkimus	23
4.1.2. Kyselytutkimus	24
4.1.3. Tutkimukseen osallistujat	25
4.1.4. Tutkimuksen luotettavuus	26
4.1.5. Tulosten käsittely ja esitys	27
4.2. Funktionaalisten ohjelmointikielten ominaisuudet	28
4.2.1. Datan käsittely	28
4.2.2. Puhtaus	32
4.3. Funktionaalisen ohjelmoinnin tehokkuus	33

4.3.1. Tuottavuus	33
4.3.2. Ilmaisuvoima	37
4.3.3. Aiemmat tutkimukset	41
4.4. Funktionaalisen ohjelmointikielen vaikutukset ohjelmistoprojektiin . .	42
4.4.1. Ekosysteemi	42
4.4.2. Työkalut	43
4.4.3. Projektinhallinta	44
5. Yhteenveto	45
Lähteet	47

LYHENTEET JA TERMIT

Algebrallinen tietotyyppi	<i>Algebraic data type</i> ; tietotyyppi, joka muodostuu muita tietotyyppejä yhdistämällä
CLR	<i>Common Language Runtime</i> ; virtuaalikone jossa C#-ohjelmat ajetaan
Ensimmäisen luokan kansalainen	<i>First-class citizen</i> ; arvo, joka voidaan sijoittaa muuttujaan ja joka voi toimia funktion parametrina tai paluuarvona
Ensimmäisen luokan funktio	<i>First-class function</i> ; funktio, joka on ensimmäisen luokan kansalainen
Hahmontunnistus	<i>Pattern matching</i> ; Menetelmä, jossa tulos valitaan jonkin arvon tietotyypin tai sisällön perusteella
Homoikonisuus	<i>Homoiconicity</i> ; periaate, jossa ohjelmakoodi esitetään ohjelman itsensä ymmärtämänä tietorakenteena
HTTP	<i>HyperText Transfer Protocol</i> ; protokolla jota käytetään web-palvelimen kanssa kommunikointiin
Häntärekursio	<i>Tail recursion</i> ; rekursion tyyppi jossa rekursiokutsun palauttama arvo palautetaan edelleen suoraan
JVM	<i>Java Virtual Machine</i> ; virtuaalikone jossa Java-ohjelmat ajetaan
Lause	<i>Statement</i> ; imperatiivisen ohjelmoinnin pienin yksikkö; ohje yksittäisen toiminnon suorittamiseksi
Lauseke	<i>Expression</i> ; arvojen ja funktioiden yhdistelmä, joka evaluoituu tietyksi arvoksi
Muuttumattomuus	<i>Immutability</i> ; periaate, jossa olion arvoa ei voi muuttaa sen luomisen jälkeen
Ohjelmointiparadigma	Ohjelmointikielen tapa mallintaa ongelma
REPL	<i>Read-Eval-Print Loop</i> ; vuorovaikutteinen tulkki, jolla voi suorittaa ohjelmakoodia ja nähdä välittömästi sen tuloksen

REST	<i>Representational State Transfer</i> ; periaate web-palvelimen rajapinnan toteutukseen. REST-rajapinta perustuu asiakas-palvelin-malliin ja on tilaton.
Rinnakkaisuus	<i>Parallelism</i> ; Suorituksen jakaminen useaan yhtä aikaa suoritettavaan osatehtävään
Rinnakkaisuus	<i>Concurrency</i> ; Samaan aikaan suoritettavien tehtävien välinen kommunikointi ja resurssien (kuten ohjelman käyttämän muistin) jakaminen
Sivuvaikutus	<i>Side effect</i> ; funktion suorittamisen aiheuttama ulkoisen tilan muutos
Sulkeuma	<i>Closure</i> ; Ensimmäisen luokan funktion tallettama tila ympäristöstä, jossa funktio määriteltiin
Vapaa muuttuja	<i>Free variable</i> ; muuttuja jonka arvoa ei ole annettu lausekkeessa vaan joka otetaan lausekkeen ulkopuolisesta kontekstista

1. JOHDANTO

Web-ohjelmoinnin suosio kasvaa jatkuvasti. Suuri osa nykypäivänä luotavista ohjelmistoista toimii ainakin osittain internetiin kytkeytyneenä, ja erilaisten verkkopalveluiden määrä on huimaava. Web-sivut itsessään ovat kehittyneet staattisista HTML-sivuista kokonaisvaltaisiksi ohjelmiksi, jotka toimivat käyttäjän toimenpiteisiin ja web-palvelinten lähettämään dataan perustuen. Verkkoselain tarjoaa kaikille laitteille yhtenäisen toiminta-alustan ja mahdollistaa saman palvelun käyttämisen kaikissa tilanteissa sekä erilaisilla päätelaitteilla. Erilaiset pilvipalvelut ovat helpottaneet web-ohjelmistojen julkaisemista. Web-ohjelma ei myöskään vaadi erillistä asentamista käyttäjän koneelle tai asennetun ohjelman päivittämistä. [1]

Myös funktionaalisen ohjelmoinnin suosio on kasvussa. Aiemmin sen käyttöä rajoittivat muisti- ja laskentakapasiteetin vähäisyys, mutta nykyään absoluuttisella suorituskyvyllä on vähemmän merkitystä. Funktionaalisen ohjelmoinnin hyödyt on havaittu, ja sen käyttö erityisesti web-ohjelmoinnissa on suurempaa kuin aiemmin.

Funktionaalisen ohjelmoinnilla on pitkä historia. Funktionaalisen ohjelmoinnin ajatus syntyi 1930-luvulla lambda-kalkyylin muodossa. Lisp, ensimmäinen funktionaalinen ohjelmointikieli, syntyi jo 1950-luvulla, ja sen eri murteet ovat tänäkin päivänä käytössä. Koska funktionaalinen ohjelmointi pohjautuu matematiikkaan eikä tietokoneen sisäiseen toimintalogiikkaan kuten imperatiivinen ohjelmointi, sen ajatusmaailma ei vanhene. Siksi periaatteet, jotka viime vuosisadan alussa johtivat funktionaalisen ohjelmoinnin syntymiseen, ovat edelleen voimassa.

Tässä diplomityössä tutkin funktionaalisen ohjelmoinnin soveltuvuutta web-ohjelmistokehitykseen. Työn lähtökohtana on Solita Oy:ssä tehty haastattelututkimus, jossa on haastateltu työntekijöitä, jotka ovat tehneet web-ohjelmistoja sekä funktionaalilla että imperatiivisilla ohjelmointikielillä. Käytössä ovat myös tulokset kyselytutkimuksesta, joka tehtiin aiemmin Solita Oy:ssä erään projektin henkilöstön kokemuksista funktionaalisen ohjelmoinnin käytöstä. Diplomityön tavoitteena

on selvittää, onko funktionaalisesta ohjelmoinnista etuja web-ohjelmistokehityksessä ja onko funktionaaliseen ohjelmointiin panostaminen yrityksen toiminnan kannalta hyödyllistä.

Tämän diplomityön luvussa 2 esittelen funktionaalisen ohjelmoinnin periaatteet sekä historian. Luvussa 3 esittelen web-ohjelmoinnin periaatteita sekä funktionaalisen ohjelmoinnin soveltamista web-ohjelmointiin. Luvussa 4 käyn läpi tutkimuksen tulokset aihepiireittäin, ja luvussa 5 esitän yhteenvedon koko työstä ja sen tuloksista. Koska monien termien suomenkieliset vastineet eivät ole täysin vakiintuneita, suomenkielisten termien englanninkieliset vastineet on esitetty suomenkielisen termin jälkeen suluissa kursiivilla. Lisäksi kyseiset termit on selitetty lyhenteiden ja termien luettelossa.

2. FUNKTIONAALINEN OHJELMOINTI

Funktionaalinen ohjelmointi on ohjelmointiparadigma, joka perustuu matemaattisen funktion käsitteeseen ja siten lausekkeiden evaluointiin. Puhdas funktionaalinen ohjelmointi kuuluu deklarativiisiin paradigmoihin, eli sillä pyritään kuvaamaan *mitä* halutaan ratkaista sen sijaan, että kuvattaisiin, *miten* ongelma halutaan ratkaista. Ohjelmointikielen sisäinen toteutus huolehtii siitä, miten ongelma käytännössä ratkaistaan, kun ohjelmoija on kuvailut, mitä haluaa saavuttaa. Puhtaassa funktionaaliossa ohjelmoinnissa ohjelmalla ei ole ollenkaan implisiittistä sisäistä tilaa, sillä funktioilla ei voi olla sivuvaikutuksia. Ainoa käytettävissä oleva tila koostuu funktioiden syötteistä ja tuloksista. Imperatiiviossa ohjelmoinnissa vastaavasti on sisäinen tila, jota muutetaan. [2]

2.1. Funktionaalisen ohjelmoinnin periaatteet

Funktionaalisen ohjelmoinnin lähtökohtana on funktioiden puhtaus. Puhtaaksi funktioksi kutsutaan funktiota, jolla ei ole sivuvaikutuksia ja jonka arvo ei riipu ohjelman tilasta. Sivuvaikutuksella tarkoitetaan muutosta, jonka funktio tekee ympäristöönsä. Sivuvaikutuksia ovat esimerkiksi muuttujan arvon muokkaaminen, tiedostoon kirjoittaminen tai käyttöliittymän päivitys. [3]

Koska sijoitusoperaation aiheuttama tilan muutos on sivuvaikutus, se ei ole puhtaassa funktiossa mahdollinen, joten puhtaassa funktionaaliossa ohjelmoinnissa ei ole lainkaan muuttujia. Tämä on suurin ero imperatiiviossa ja funktionaaliossa ohjelmoinnin välillä. Puhtaassa funktionaaliossa ohjelmoinnissa kaikki tietotyypit ja tietorakenteet ovatkin muuttumattomia (*immutable*). Muuttumattomuus tarkoittaa, ettei rakennetta voi muuttaa sen jälkeen, kun se on luotu.

Muuttujien puuttumisesta seuraa, että monet perusasiat tehdään funktionaaliossa kielissä eri tavoin kuin imperatiiviossa kielissä. Silmukka, jonka loppuminen ta-

pahtuu muuttujan arvon perusteella, ei ole mahdollinen. Sen sijaan funktionaalisisessa kielessä käytetään rekursiivista funktiota. [2] Ohjelmassa 2.1 on esitetty kokonaisluvun kertoman laskeminen imperatiivisesti käyttäen muuttujia ja silmukkaa. Ohjelmassa 2.2 sama on tehty Clojurella rekursiivisesti. Silmukan sijaan määritellään funktion arvo, kun parametrinä on 0, ja muissa tapauksissa kutsutaan rekursiivisesti samaa funktiota yhtä pienemmällä parametrillä.

Ohjelma 2.1: Kertomafunktio Javalla silmukan avulla

```
1 public int factorial(int n) {
2     int result = 1;
3     while (n > 0) {
4         result = result * n;
5         n = n - 1;
6     }
7     return result;
8 }
```

Ohjelma 2.2: Kertomafunktio Clojurella

```
1 (defn factorial [n]
2   (if (zero? n)
3       1
4       (* n (factorial (dec n)))))
```

Rekursioiden erikoistapauksena on häntärekursio (*tail recursion*), jossa rekursiokutsu on viimeinen kutsujan tekemä toimenpide. Häntärekursiosta on etua ohjelmaa suorittaessa, koska kääntäjä voi optimoida funktiokutsun kokonaan pois käännetystä ohjelmasta. [2] Ohjelmassa 2.3 on esitetty kertomafunktio häntärekursiivisena. Ensimmäinen määritellään kahden parametrin apufunktio, jossa varsinainen rekursio tapahtuu. Funktion toisena parametrina välitetään laskennan senhetkinen tulos, ja kuten ohjelmakoodista nähdään, rivillä 5 tapahtuvan rekursiokutsun tulos palautetaan suoraan. Clojuressa häntärekursio tapahtuu poikkeuksellisesti `recur`-kutsulla eikä rekursiivisen funktion nimellä johtuen JVM-alustan rajoitteista. Itse 1-parametrinen `factorial`-funktio vain kutsuu kahden parametrin rekursiivista apufunktiota.

JVM-alustalla tukea häntärekursiolle ei ole. JVM-pohjaisten kielten kääntäjien täytyy siis itse toteuttaa häntärekursion purkaminen silmukaksi. [4] Clojuressa tämä tapahtuu edellämainitulla `recur`-kutsulla. Koska kaikki JVM-tavukoodin hyppäskyt (mukaanlukien `goto`) sallivat hyppäämisen vain saman metodin sisällä [5], on useamman funktion keskinäisen häntärekursion (*mutual tail recursion*) purkaminen mahdotonta sellaisenaan. Keskinäiseen häntärekursioon täytyy JVM-kielissä käyttää muita työkaluja. [4][6] JVM-pohjaisista kielistä ainakin Scalassa yksittäisen funktion häntärekursion optimointi tapahtuu automaattisesti. [7]

Ohjelma 2.3: Kertomafunktio Clojurella häntärekursiivisesti

```

1 (defn factorial [n]
2   (let [factorial-recursive (fn [n result]
3                               (if (zero? n)
4                                   result
5                                   (recur (dec n) (* n result))))]
6     (factorial-recursive n 1)))

```

Funktioiden puhtaudesta seuraa useita etuja. Koska funktion arvo riippuu ainoastaan sen syötteestä, on funktiolla aina sama arvo samalla syötteellä, mikä tunnetaan nimellä viittausten läpinäkyvyys (*referential transparency*). Se mahdollistaa esimerkiksi toistuvasti suoritettavan funktion tulosten tallentamisen. Peräkkäin suoritettavien funktioiden suorituksen pystyy muuttamaan rinnakkaiseksi, koska ne eivät voi vaikuttaa toistensa toimintaan. Puhtaan funktion arvon voi myös laskea laiskasti vasta sitten, kun sitä tarvitaan. Epäpuhtaalla funktiolla sama ei olisi mahdollista, koska funktion sivuvaikutukset tapahtuisivat ennalta määrittelemättömänä ajanhetkenä. [8]

Koska funktionaalisen ohjelmoinnin peruskonsepti on funktio, on luonnollista, että funktiot ovat niissä arvoja, joita voidaan käsitellä muiden arvojen tapaan. Tällaisesta arvosta, jonka voi sijoittaa muuttujaan ja joka voi toimia funktion parametrina tai paluuarvona, käytetään nimitystä ensimmäisen luokan kansalainen (*first-class citizen*). Useissa imperatiivisissa kielissä funktiot eivät ole ensimmäisen luokan kansalaisia. Funktioista ensimmäisen luokan kansalaisina käytetään joskus myös termiä ensimmäisen luokan funktio (*first-class function*).

Yleinen ensimmäisen luokan funktioihin liittyvä ominaisuus funktionaalisisissa kielessä on korkeamman asteen funktio, millä tarkoitetaan funktiota, joka ottaa parametrikseen toisen funktion tai palauttaa funktion paluuarvonaan. Hyvä esimerkki korkeamman asteen funktiosta on `map`, joka ottaa parametrikseen funktion sekä listan, suorittaa kyseisen funktion jokaiselle listan alkioille ja palauttaa sen paluuarvoista muodostetun uuden listan. Toinen esimerkki on funktiokompositio (*function composition*). Sillä tarkoitetaan funktiota, joka ottaa parametrikseen kaksi funktiota ja palauttaa uuden funktion, joka on parametrinä annettujen funktioiden yhdistelmä. Matematiikassa sama tunnetaan yhdistettynä funktiona. [8]

Koska ensimmäisen luokan funktion suorituspaikkaa tai -hetkeä ei tiedetä etukäteen, täytyy niiden tallettaa tietoa ympäristöstään. Tätä kutsutaan sulkeumaksi (*closure*). Sulkeuma tallettaa kaikkien funktion vapaiden muuttujien arvot siitä ympäristöstä, missä funktio määritellään. Ohjelmassa 2.4 on esitetty sulkeuman toiminta. Funktio `adder` palauttaa toisen funktion, joka lisää parametriinsa luvun `x`. Palautettu funktio ei saa `x`:ää parametrinään, vaan `x:n` arvo tallennetaan sulkeumana, kun funktio luodaan `adder`-funktion suorituksen aikana. Niinpä `(adder 3)` palauttaa funktion, joka lisää parametriinsä luvun 3.

Ohjelma 2.4: Sulkeuma Clojure-funktiossa

```

1 (defn adder [x]
2   (fn [y]
3     (+ x y)))
4
5 (def add3 (adder 3))
6
7 (add3 5) ;; => 8
```

2.2. Funktionaalisen ohjelmoinnin historia

2.2.1. Lambdakalkyyli

Lambdakalkyyli on Alonzo Churchin vuonna 1932 julkaisema[9] teoreettinen laskentamalli, jonka tavoitteena oli muodostaa uusi perusta matemaattiselle logiikalle.

Church muodosti lambdakalkyylin funktion käsitteelle, kun aiemmat teorit olivat yleensä perustuneet matemaattisiin joukkoihin. Churchin alkuperäinen teoria oli kuitenkin osittain inkonsistentti, joten Church rajasi teoriastaan konsistentin osuuden, joka nykyään tunnetaan tyypittömänä lambdakalkyylinä. [10]

Lambdakalkyyli muodostuu niin kutsutuista lambdalausekkeista, jotka on määriteltä seuraavasti: [11]

1. Pelkkä muuttuja x on lambdalauseke.
2. Kun e on lambdalauseke ja x muuttuja, $\lambda x.e$ on lambdalauseke. Tämä sääntö on nimeltään abstraktio (*abstraction*). Tässä lausekkeessa x on sidottu muuttuja.
3. Kun d ja e ovat lambdalausekkeita, $(d e)$ on lambdalauseke. Tämä sääntö on nimeltään applikaatio (*application*), ja tarkoittaa funktion d kutsumista argumentilla e .

Lambdalausekkeiden merkitys pohjautuu myös kolmeen sääntöön: [11]

1. Sidottujen muuttujien nimellä ei ole väliä: $\lambda x.x$ ja $\lambda y.y$ ovat sama asia. Tämä sääntö on nimeltään α -konversio (*α -conversion*).
2. Applikaatio tapahtuu korvaamalla funktion sidottu muuttuja sen argumentilla: $(\lambda x.e e')$ on $e[x := e']$, eli jokainen x e :ssä korvataan e' :lla. Tämä sääntö on nimeltään β -reduktio (*β -reduction*).
3. Kaksi funktiota on sama asia, jos ja vain jos niillä on kaikilla argumenteilla sama lopputulos: $\lambda x.(f x)$ ja f ovat sama asia. Tämä sääntö on nimeltään η -konversio (*η -conversion*).

Kuten tästä määritelmästä nähdään, ainoa lambdakalkyyllissä olemassaoleva tyyppi on funktio. Kaikki muut tyypit onkin määriteltävä funktioiden pohjalta. Esimerkiksi kokonaislukujen esittämiseen ja laskutoimituksiin voidaan määritellä tietynlaiset funktiot. Tunnetuin tapa esittää kokonaisluvut tunnetaan nimellä Churchin numeraalit (*Church numerals*). [11]

2.2.2. Lisp

Varsinainen funktionaalinen ohjelmointi syntyi 1950-luvulla, kun John McCarthy kehitti Lisp-kielen Massachusetts Institute of Technologyssa [12]. Vaikka Lisp ei suoraan perustunutkaan lambdakalkyyliin, McCarthy otti siitä vaikutteita kieleensä: esimerkiksi nimettömän funktion syntaksi Lispissä on $(\lambda (x) e)$, mikä vastaa lambdakalkyylin lauseketta $\lambda x.e$.

McCarthy tavoitte oli kehittää kieli tekoälytutkimusta varten. Lisp pohjautui listojen käsittelylle, mistä sen nimikin on peräisin (LIST Processing). Ainoa olemassaoleva korkeamman tason kieli oli imperatiivinen FORTRAN, joten sen ja (myös imperatiivisen) konekielisen ohjelmoinnin jälkeen funktionaalista kieltä pidettiin melko radikaalina.

Lisp sisälsi useita ominaisuuksia, jotka nykyäänkin vielä määrittelevät funktionaalisia kieliä: ehtolauseke, jonka avulla pystyi toteuttamaan todellisen rekursion; listojen käyttö ja niiden käsittely korkeamman asteen funktioilla sekä linkitettyjen listojen toteutus ns. `cons`-soluna. Myös ohjelmakoodin koostaminen pelkistä lausekkeista (*expression*) lausekkeiden ja lauseiden (*statement*) sijaan sekä automaattinen roskienkeruu ovat lähtöisin Lispistä. Lisp ei kuitenkaan vielä ollut puhdas funktionaalinen kieli, vaan sisälsi useita imperatiivisia ominaisuuksia. Vasta myöhemmät Lisp-murteet, kuten Scheme, ovat siirtyneet puhtaampaan suuntaan ja lähemmäs lambdakalkyyliä. [2]

Merkittävänä erona Lispin ja muiden ohjelmointikielten välillä on se, että Lisp käyttää samaa syntaksia sekä ohjelmakoodin että datan esittämiseen. Kaikki Lisp-koodi on siis myös puurakenteena olevaa dataa. Tämä ominaisuus tunnetaan nimellä homoikonisuus (*homoiconicity*). Lispin syntaksi perustuu niinkutsuttuihin S-lausekkeisiin (*S-expression*). S-lausekkeet muodostavat puurakenteen, joka koostuu sisäkkäisistä listoista. Yksittäinen lista puolestaan koostuu suluilla ympäröidyistä arvoista. Esimerkiksi $(x\ y\ (a\ b\ c))$ on S-lauseke. Funktiokutsu esitetään Lispissä S-lausekkeena, jonka ensimmäinen arvo on kutsuttava funktio ja loput arvot kyseisen funktion parametrit. Esimerkiksi laskutoimitus $1+2$ esitetään Lispissä muodossa $(+ 1 2)$. [2]

Homoikonisuus mahdollistaa erittäin voimakkaan makrojärjestelmän olemassaolon. Lisp-makrot ovat funktioita, jotka ottavat syötteekseen koodia, muokkaavat sitä ja palauttavat muokatun koodin, jota käytetään lopullisen ohjelman osana. Koodin muokkaus tällä tasolla on mahdollista ainoastaan, koska kaikki ohjelmakoodi on käsiteltävissä olevaa dataa. Lispiä on tämän vuoksi kutsuttu "ohjelmoitavaksi ohjelmointikieleksi". [13]

Koko Lispin olemassaolo toimivana ohjelmointikielenä perustuu samaan asiaan. John McCarthy oli alun perin suunnitellut Lispin ainoastaan teoreettiseksi kieleksi todistaakseen, että matemaattisista funktioista ja algoritmeista syntyy Turing-täydellinen kieli. McCarthy'n oppilas Steve Russell kuitenkin huomasi, että toteuttamalla pelkän `eval`-funktion konekielellä syntyy toimiva Lisp-tulkki. Funktio `eval` laskee Lisp-lausekkeen arvon, ja sitä käyttämällä pystyy siis suorittamaan kaiken Lisp-koodin. Russell saikin tehtyä ensimmäisen todellisen toteutuksen Lisp-kielelle. [14]

Lisp lähti julkaisunsa jälkeen kehittymään useaan suuntaan. Tärkeimmät Lisp-murteet olivat Scheme ja Common Lisp, joista molemmilla oli erilaiset lähtökohdat ja lähestymistavat Lispiin ja funktionaaliseen ohjelmointiin. Common Lisp pyrki laajentamaan Lispiä, ja sisälsi huomattavan määrän erilaisia ominaisuuksia. Scheme sitä vastoin pyrki minimaalisuuteen ja puhtauteen. Niiden lisäksi vähemmän tunnettuja ja käytettyjä Lisp-murteita on lukematon määrä. Uusin laajempaan käyttöön päässyt Lisp-murre on vuonna 2007 julkaistu Clojure.

Yksi syy Lisp-murteiden suurelle määrälle on se, että Lisp-kielen luominen on pohjimmiltaan huomattavasti helpompaa kuin monen muun kielen. Koko kielen toiminta vaatii vain lukijan, joka muodostaa ohjelmakoodista tietorakenteen, sekä `eval`-funktion, joka laskee lausekkeiden arvot. Esimerkiksi monimutkaista kääntäjää ei tarvita laisinkaan. Sanotaan myös, että Lisp ei ole vanhentunut, koska se ei ole teknologiaa vaan matematiikkaa. Siinä mielessä se on ehkä lähimpänä funktionaalisen ohjelmoinnin olemusta. [14]

2.2.3. ML

Robin Milner kehitti ML-kielen 1970-luvulla Edinburghin yliopistossa. ML on staattisesti tyyppitetty funktionaalinen ohjelmointikieli, joka luotiin teoreemien todistamisen apuvälineeksi. Todistamiseen käytettiin järjestelmää nimeltä LCF (*Logic for Computable Functions*), ja se oli alun perin toteutettu Lispillä. Milner kuitenkin halusi todistamisen apuvälineeksi staattisen tyyppityksen. Tätä varten hän kehitti LCF:n toteutukseen uuden ohjelmointikielen, ML:n. ML on lyhenne sanoista *Meta Language*. [15]

ML-kielen tunnetuin perintö on sen tyyppipäättelyjärjestelmä. J. Roger Hindley oli aiemmin luonut algoritmin tyyppitetyn kombinaatiologiikan tyyppipäättelylle. [16] Milner löysi Hindleyn algoritmin, ja teki ML:ään sen pohjalta tyyppipäättelyn. Hindley–Milner-järjestelmäksi nimetty algoritmi ei vaadi ollenkaan ohjelmoijan merkitsemiä tyyppejä, ja pystyy löytämään yleisimmän mahdollisen tyyppin annetulle ohjelmalle. Selkeyden vuoksi tyyppien merkitseminen näkyviin on kuitenkin sallittua, muttei pakollista. [17]

Toinen ML-kielestä peräisin oleva, funktionaalisissa kielissä yleinen ominaisuus on hahmontunnistus (*pattern matching*), sekä siihen vahvasti liittyvät algebralliset tietotyypit (*algebraic data types*). Hahmontunnistuksella tarkoitetaan, että arvo sovitetaan esimerkiksi tietotyypin tai sisällön perusteella yhteen useista vaihtoehdoista, jonka mukaan lausekkeen arvo määräytyy. Algebrallisilla tietotyypeillä tarkoitetaan tietotyyppejä, joita voidaan yhdistellä uusiksi tyypeiksi. [2]

Ohjelmassa 2.5 on esitetty algebrallisten tietotyyppien toimintaa. Ohjelman syntaksi on Haskellia, joka on ML:ään pohjautuva funktionaalinen ohjelmointikieli ja jonka syntaksi on lähellä ML:n syntaksia. Ohjelmassa esitellään kaksi algebrallista tietotyyppiä. Ensimmäinen niistä on `Maybe a`, joka koostuu tyypeistä `Nothing` ja `Just a`. `Nothing` kuvaa arvon puuttumista ja `Just a` tyyppin `a` arvoa. `Maybe a` kuvaa siis tyyppin `a` arvoa, jonka olemassaolo ei ole varmaa. Toinen tietotyyppi on `List a`, joka koostuu tyypeistä `Nil` sekä `Cons`. `Nil` kuvaa tyhjää listaa, ja `Cons` listaa jossa on alkio tyyppiä `a` sekä listan loppuosa tyyppiä `List a`. Sen jälkeen esitellään hahmontunnistus näille molemmille. Kummassakin tapauksessa kyseisen tyyppin arvoa vertaillaan eri vaihtoehtoihin, ja valitaan se vaihtoehto, jonka tyyppi täsmää.

Hahmontunnistusta voidaan käyttää myös funktion määrittelyyn: ohjelmassa määritellään kertomafunktio hahmontunnistuksen avulla. Parametria verrataan arvoihin 0 ja n (joka kuvastaa mitä tahansa muuta arvoa) ja valitaan täsmäävä lopputulos. Tässä tapauksessa vertailukohteena toimii annetun arvon tyyppin sijaan arvo itse.

Ohjelma 2.5: Algebralliset tietotyypit ja hahmontunnistus

```

1 data Maybe a = Nothing | Just a
2 data List a = Nil | Cons a (List a)
3
4 case x of
5   Just value -> ...
6   Nothing    -> ...
7
8 case list of
9   Nil          -> ...
10  x :: xs -> ...

```

Ohjelma 2.6: Kertomafunktio hahmontunnistuksen avulla

```

1
2 factorial :: Integer -> Integer
3 factorial 0 = 1
4 factorial n = n * factorial (n - 1)

```

ML-kielestä on edelleen kehitetty monia versioita, joista tunnetuimmat ovat Standard ML ja Caml (sekä sen olio-ohjelmointiin suunnattu versio OCaml). Myös Haskell ja F# ovat ottaneet vahvasti vaikutteita ML-kielestä.

2.3. Funktionaalinen ohjelmointi nykyään

2.3.1. Clojure ja ClojureScript

Clojure on Rich Hickeyn vuonna 2007 kehittämä funktionaalinen ohjelmointikieli. Se on Lisp-kielen murre, joka käännetään JVM:ssä (*Java Virtual Machine*)suoritettavaksi. Clojure-ohjelmat ajetaan siis samalla alustalla kuin Java-ohjelmatkin, minkä vuoksi Clojure on tehty täysin yhteensopivaksi Javan kanssa. Clojure-ohjelmasta pystyy kutsumaan myös Java-koodia, joten kaikki Java-kirjastot

ovat myös Clojure-ohjelmien käytettävissä. Clojuresta on olemassa myös CLR:lle käännettävä versio, mutta se ei ole saavuttanut läheskään yhtä suurta suosiota kuin JVM-pohjainen versio. ClojureCLR:ssä pystyy kutsumaan C#-koodia vastaavasti kuin Clojuressa Java-koodia.

Muiden Lisp-murteiden tavoin Clojure on dynaamisesti tyyppitetty kieli. Sivuvaikutuksia ei ole kielen tasolla estetty, joten funktioiden puhtaus on ohjelmoijan vastuulla. Lisp-murteena Clojuressa on myös voimakas makrojärjestelmä.

Clojuren ominaisuuksiin kuuluvat myös muuttumattomat, persistentit tietorakenteet (*immutable, persistent*). Persistentillä tietorakenteella tarkoitetaan tietorakennetta, joka säilyttää aikaisemmat versionsa ja jakaa tilan niiden kanssa. Esimerkiksi jos vektoriin `[1 2 3]` lisätään alkio `4`, alkuperäinen vektori ei muutu, ja uusi vektori sisältää ainoastaan viittauksen vanhaan ja lisätyn alkion `4`, jolloin lopputuloksena on `[1 2 3 4]`. Kaikki muutkin Clojuren tietotyypit ovat muuttumattomia, mikä kannustaa puhtaaseen funktionaaliseen ohjelmointiin.

ClojureScript on Clojuren versio, joka käännetään JVM-tavukoodin sijaan JavaScriptiksi. Tällöin sitä voidaan suorittaa selaimessa, mikä mahdollistaa käyttöliittymäkoodin tekemisen Clojurella. ClojureScriptin ja Clojuren yhdistämällä sekä palvelin- että käyttöliittymäohjelmisto voidaan tehdä samalla ohjelmointikielellä, mikä vähentää ohjelmakokonaisuuden monimutkaisuutta ja mahdollistaa saman ohjelmakoodin käyttämisen sekä palvelin- että käyttöliittymäohjelmassa.

2.3.2. Scala

Scala on vuonna 2004 julkaistu funktionaalinen ohjelmointikieli. Kuten Clojure, sekin käännetään JVM:ssä suoritettavaksi tavukoodiksi ja on täysin yhteensopiva Javan kanssa. Toisin kuin Clojure, Scala on staattisesti tyyppitetty kieli, jossa on myös tehokas tyyppipäättely. Scalaan on otettu vaikutteita Lispin sijaan pääosin Haskellista ja ML:stä, ja siinä on paljon näistä kielistä lainattuja ominaisuuksia kuten tyyppipäättely, hahmontunnistus ja korkeamman luokan funktiot. Scalan oliojärjestelmään puolestaan on otettu vaikutteita Javasta ja Smalltalkista. [18] Scala pyrkii yhdistämään imperatiivisen olio-ohjelmoinnin sekä funktionaalisen ohjelmoinnin, ja tarjoaa mahdollisuudet kumpaankin.

2.3.3. Java 8

Kuten moniin muihinkin ohjelmointikieliin nykyään, myös Javaan lainataan ominaisuuksia funktionaalista ohjelmointikielistä. Suurimmat Javan versioon 8 tulleet uudistukset olivat Stream API sekä lambda-lausekkeet.

Stream API tarjoaa työkalut listojen käsittelyyn funktionaalaisella, deklaratiivisella tavalla. Se toteuttaa esimerkiksi monista muista kielistä tutut `map`- sekä `filter`-funktiot. Lambda-lausekkeet tuovat Javaan anonyymit funktiot. Koska Javassa funktiot eivät kuitenkaan ole ensimmäisen luokan kansalaisia, lambda-lausekkeet eivät tosiasiassa toteuta funktiota. Sen sijaan, lambda-lausekkeella tehdään toteutus rajapinnasta, jolla on täsmälleen yksi metodi. Ohjelmassa 2.7 on esitetty tällaisen rajapinnan toteutus sekä Java 7:lla että Java 8:n lambda-lausekkeella. Ohjelman rivillä 20 oleva lambda-lauseke toteuttaa `Predicate`-rajapinnassa määritellyn `test`-metodin ja on identtinen riveillä 13–17 tehdyn rajapintatoteutuksen kanssa.

Ohjelma 2.7: Lambda-lausekkeella toteutettava rajapinta

```
1 @FunctionalInterface
2 public interface Predicate<T> {
3     boolean test(T t);
4 }
5
6 public interface Stream<T> {
7     Stream<T> filter(Predicate<T> predicate);
8 }
9
10 // Toteutetaan Predicate, joka palauttaa parillisille luvuille true
11
12 // Predicate-rajapinnan toteutus Java 7:lla
13 stream.filter(new Predicate<Integer> {
14     public boolean test(Integer i) {
15         return i % 2 == 0;
16     }
17 }
18
19 // Predicate-rajapinnan toteutus lambda-lausekkeella
20 stream.filter(i => i % 2 == 0);
```

2.3.4. F#

F# on Microsoftin vuonna 2005 julkaisema funktionaalinen ohjelmointikieli. Vastavasti kuin Clojure JVM:lle, F# on tehty C#:n alustalle CLR:lle. Samoin se on täysin yhteensopiva muiden saman alustan ohjelmointikielten kanssa, erityisesti C#:n.

F#:n alkuperäinen ajatus oli toteuttaa ML-tyyppinen kieli CLR:n päälle. Lähtökohdiana toteutukselle oli erityisesti OCaml. F#-koodi onkin hyvin samankaltaista kuin OCaml-koodi, vaikka F#:iin on kerätty vaikutteita ja ominaisuuksia myös muista funktionaalisisista kielistä, kuten Haskellista. [19]

2.3.5. Haskell

Vuonna 1987 *Functional Programming Languages and Computer Architecture* -konferenssissa päätettiin kehittää uusi, avoimen standardin funktionaalinen ohjel-

mointikieli tutkimuskäyttöä varten. Olemassa oli toistakymmentä funktionaalista, laiskaan evaluointiin perustuvaa ohjelmointikieltä, mutta akateemiseen tutkimukseen haluttiin saada yksi avoin standardi, jonka kautta tutkimusta voitaisiin viedä yhteiseltä pohjalta eteenpäin. Tästä lähtökohdasta syntyi Haskell. [20]

Haskell on staattisesti tyyppitetty, laiskasti evaluoitu funktionaalinen ohjelmointikieli. Kuten ML:ssä, myös siinä on käytössä Hindley–Milner-tyyppipäätely. Sen syntaksi on myös lähellä ML:ää. Suurena erona kuitenkin on, että Haskell on lähtökohtaisesti täysin puhdas. Sivuvaikutukset on sallittu ainoastaan erillisessä, niitä varten tarkoitettussa rakenteessa.

Koska Haskell oli alunperinkin suunniteltu tutkimuskäyttöön, suurin osa sen käyttökohteista on yliopistomaailmassa sekä opetuksen että tutkimuksen puolella. Haskellia on siitä huolimatta käytetty myös yritysmaailmassa.

3. WEB-OHJELMOINTI

Web-ohjelmoinnilla tarkoitetaan sellaisten ohjelmien tekemistä, joita käytetään web-selaimen kautta. Tyypillinen ohjelma koostuu kolmesta osasta: käyttöliittymästä, joka on selaimessa toimiva web-sivu; palvelinohjelmasta, jonka kautta käyttöliittymä ladataan ja johon käyttöliittymä on yhteydessä sekä tietokannasta, johon ohjelman data tallennetaan. Yleisimmin web-ohjelmassa on yksi tietokanta, yksi palvelin sekä useita käyttäjiä, joista jokaisella on selaimessaan oma käyttöliittymä.

Web-ohjelmassa käyttöliittymä ja palvelin kommunikoivat keskenään yleensä HTTP-protokollan välityksellä. HTTP-protokollassa asiakas eli käyttöliittymä lähettää palvelimelle pyyntöjä, joihin palvelin vastaa. Nämä pyynnöt voivat olla esimerkiksi tiedon hakua tai tallentamista. Jokainen pyyntö kohdistuu tiettyyn URL-osoitteeseen. Pyyntöillä on tyyppi, kuten `GET` (tiedonhaku), `POST` (tiedon tallentaminen), `PUT` (tiedon päivittäminen) tai `DELETE` (tiedon poistaminen). Tallennus- ja päivityspyyntöön voi myös sisältyä dataa.

3.1. Web-palvelin

Web-palvelin on ohjelma, joka vastaanottaa HTTP-pyyntöjä ja lähettää niihin vastauksia. Yksinkertaisimmillaan palvelin vain tarjoilee staattisia tiedostoja, mutta yleensä se on laajempi ohjelma, joka voi liittyä edelleen muihin ohjelmiin. Yleisin liittymä web-palvelimella on tietokanta, jossa palvelin säilyttää dataa.

Palvelinohjelma pyritään pitämään tilattomana. Kaikki käyttäjäkohtainen tila säilytetään käyttöliittymässä ja palvelimen kanssa kommunikoitaessa tarvittava tila sisällytetään HTTP-pyyntöön esimerkiksi evästeen muodossa. Tämä periaate tunnetaan termillä REST, ja se on hyvin usein pohjana palvelinohjelman toteutuksessa, vaikkei sitä yleensä toteuteta täydellisesti. REST-periaatteen mukainen rajapinta perustuu asiakas-palvelin-malliin, jossa palvelin on tilaton. Palvelin tarjoaa

asiakkaille yhdenmukaisen rajapinnan, ja toistuvat pyynnöt voidaan tallentaa välimuistiin. Asiakkaita ei ole rajattu pelkästään web-selaimen kautta toimivaan käyttöliittymään, vaan rajapintoja voidaan käyttää myös muiden ohjelmien kautta, eikä käyttöön välttämättä liity edes ihmiskäyttäjää, vaan asiakas voi olla myös täysin ohjelmallinen. [21]

Palvelinohjelma vastaa myös ohjelmaan liittyvän datan tallentamisesta tietokantaan. Usein palvelinohjelma toimiikin vain rajapintana käyttöliittymän ja tietokannan välissä, ja palvelimen vastuulle kuuluu datan muuttaminen käyttöliittymän ymmärtämään muotoon ja pääsyn rajaaminen tietokantaan. Palvelin myös hoitaa tarvittaessa käyttäjien tunnistamisen ja valtuuttamisen (*authentication, authorization*).

3.2. Web-käyttöliittymä

Webin alkuaikoina sivustot koostuivat ainoastaan staattisista HTML-sivuista. Ainoa käyttäjälle tarjottu interaktiomahdollisuus oli sivulta toiselle siirtyminen hyperlinkkien avulla. Sivujen sisältö oli kaikille käyttäjille sama, ja sisältöä päivitettiin muokkaamalla palvelimella olevia HTML-tiedostoja.

Myöhemmin interaktion mahdollisuudet kasvoivat. Käyttäjän täyttämät lomakkeet, niiden käsittely, datan tallennus tietokantaan ja palvelimella ajettavat ohjelmat toivat mahdollisuuden dynaamisten sivujen luomiseen. Sisällön generoiva ohjelma suoritettiin kuitenkin edelleen palvelimella, ja käyttäjä sai selaimensa pelkkää HTML:ää.

Vuonna 1995 Netscape-selaimen kehittäjät julkaisivat JavaScriptin. Netscape-yhtiö ymmärsi, että verkkoselain voisi toimia ohjelmien alustana eikä pelkkänä staattisen tiedon näyttäjänä. Niinpä yhtiö loi verkkoselaimessa ajettavaksi tarkoitetun kielen, JavaScriptin. Nimensä mukaisesti se otti vahvasti vaikutteita Javasta, mutta myös muista sen ajan ohjelmointikielistä kuten C++:sta ja Schemestä. [22]

Kun Microsoft kehitti vuonna 1999 AJAX-tekniikan, siirtyi web-ohjelmointi entistä vahvemmin kohti käyttäjän selainta, kun myös datan asynkroninen lataaminen palvelimelta tuli mahdolliseksi. Ohjelman suorituksen pystyi siirtämään osittain selaimessa suoritettavaksi, kun selainohjelma pystyi pyytämään palvelimelta vain

tarvitsemansa datan eikä tarvinnut koko tietokantaa käyttöönsä. Samoin tiedon tallentaminen pystyttiin tekemään ilman kokonaisia sivulatauksia. Näin web-ohjelmat pystyivät toimimaan perinteisten ohjelmien tapaan ilman keskeytyksiä. [23]

Nykyään Web toimii jo pääasiallisena ympäristönä tietokoneohjelmistoille. Web-ohjelmiston etuihin kuuluu verkkoselainten tarjoama yhtenäinen toiminta-alusta kaikilla päätelaitteilla ja käytön mahdollisuus ajasta ja paikasta riippumatta. Ohjelmiston julkaisu ja asennus on yksinkertaisempaa sekä käyttäjän että kehittäjän näkökulmasta: käyttäjällä on aina tarjolla uusin versio ohjelmistosta ilman erillisiä asennuksia tai päivityksiä, ja ohjelmiston kehittäjä saa yhdellä toimenpiteellä uusimman version tarjolle kaikille ohjelmiston käyttäjille. [1]

Nykyisin selaimessa ajettavat ohjelmat on viety niin pitkälle, että monet verkkosivut ovat ns. yhden sivun ohjelmia (*Single Page Application, SPA*). Niissä palvelimelta ladataan ainoastaan minimaalinen sivurunko sekä JavaScript-ohjelma. JavaScript-ohjelma lataa edelleen tarvitsemansa resurssit palvelimelta, välittää käyttäjän toimet palvelimelle ja huolehtii sivulta toiselle siirtymistä sekä datan näyttämisestä. Yhden sivun ohjelmassa entistä suurempi osa ohjelman suorituksesta siirtyy käyttäjän selaimen, mikä vähentää palvelimen rasitusta ja nopeuttaa ohjelman suoritusta käyttäjän näkökulmasta.

Graafisen käyttöliittymän toiminnallisuus perustuu suurelta osin siihen, että ohjelma reagoi erilaisiin tapahtumiin. Tällaisia tapahtumia web-ohjelmistossa ovat esimerkiksi käyttäjän suorittamat toiminnot tai palvelimelta saatu vastaus pyyntöön. Yksittäisellä tapahtumalla voi olla useita seurauksia, esimerkiksi näkymän päivittäminen tai tiedon tallentaminen. Tapahtumien ja niiden seurauksien riippuvuussuhteet voivat kasvaa hyvinkin monimutkaisiksi. Sitä varten on kehitetty useita malleja, joista yleisimmät ovat takaisinkutsut (*callback*) ja tarkkailijamalli (*observer pattern*). Näistä kumpikin on yksi tapa toteuttaa reaktiivinen ohjelmointi (*reactive programming*). [24] Reaktiivisuudella tarkoitetaan sitä, että ohjelman suoritus odottaa erilaisia tapahtumia ja reagoi niihin. [3]

Takaisinkutsussa tapahtumien käsittely hoidetaan siten, että tapahtuman lähteelle annetaan parametriksi takaisinkutsufunktio, jota kutsutaan tapahtuman yhteydessä. Takaisinkutsufunktio saa parametrikseen tapahtuman tiedot, ja näin tapahtu-

masta kiinnostunut taho saa tiedon tapahtumasta. Takaisinkutsuomallissa useamman toisistaan riippuvan tapahtuman käsittely johtaa moniin sisäkkäisiin takaisinkutsuihin, mikä tekee ohjelman rakenteesta ja suorituksen kulusta erittäin monimutkaisen ja epäselvän. Kyseinen tilanne tunnetaan nimellä *takaisinkutsuhelveti* (*callback hell*). [3]

Tarkkailijamalli on olio-ohjelmoinnin puolella vastaavatyypinen suunnittelumalli. Se perustuu siihen, että tapahtuman käsittelijä toteuttaa tapahtuman lähteen määrittelemän rajapinnan, jonka kautta tapahtuma välitetään kaikille käsittelijöille. Jokainen komponentti, joka haluaa käsitellä tietyn lähteen tapahtumia, joutuu rekisteröitymään kyseisen lähteen tarkkailijaksi. Tapahtumien lähde joutuu siis pitämään kirjaa kaikista tarkkailijoista ja kutsumaan niiden rajapintaa tapahtumahetkellä. [3]

Molemmista näistä malleista on omat ongelmansa. Kummassakin tapauksessa tapahtuman käsittely perustuu sivuvaikutuksiin. Tarkkailijamalli myös aiheuttaa tarpeettoman paljon riippuvuuksia eri komponenttien välille.

3.3. Funktionaalinen web-ohjelmointi

Palvelimen vastaanottamat pyynnöt ja asiakkaan saamat vastaukset muistuttavat huomattavasti funktiokutsua ja funktion paluuarvoa. Web-ohjelman palvelinta voidaan ajatella isona funktiona, jonka syöte on pyynnön osoite ja data, ja joka palauttaa niihin perustuen arvon. Puhtaasta funktiosta on harvoin kyse, sillä pyynnön yhteydessä yleensä haetaan dataa tietokannasta tai tallennetaan dataa tietokantaan. Kumpikin näistä on sivuvaikutus. Funktio-ajattelusta on siitä huolimatta etua, sillä palvelinohjelman toiminnallisuus voidaan erottaa HTTP-kutsujen käsittelystä. Tällä tekniikalla pystytään esimerkiksi testaamaan koko palvelimen toiminnallisuutta ilman, että samalla tarvitsee luoda HTTP-palvelinta jonka kautta kutsut tehdään. Samoin varsinaisen palvelinohjelmiston vaihtaminen on helppoa, kun palvelimen toimintalogiikka ei riipu siitä. Samaa toimintalogiikkaa voisi käyttää esimerkiksi yrityksen sisäisessä testauksessa yksinkertaisella, kevyellä palvelinohjelmalla ja tuotantokäytössä järeämmällä sovelluspalvelimella.

Web-palvelimen toiminta on myös luonnostaan rinnakkaista. Kun yhtäaikaista käyttäjiä voi olla mielivaltaisen määrä, täytyy palvelimen myös kyetä palvelemaan

useampaa pyyntöä yhtä aikaa. Monet funktionaalisen ohjelmoinnin periaatteista vähentävät rinnakkaisuuden¹ mukanaan tuomia ongelmia. Funktioiden puhtaus sekä datan muuttumattomuus estävät kilpailutilanteiden (*race condition*) syntymistä, eikä ohjelmoijan tarvitse pelätä toisen samaan aikaan palveltavan pyynnön aiheuttamia sivuvaikutuksia.

Web-käyttöliittymässä funktionaaliseen ohjelmointiin liittyy yleensä kaksi ongelmaa. Web-käyttöliittymällä on lähes aina sisäinen tila, mikä ei sovellu hyvin puhtaaseen funktionaaliseen ohjelmointiin. Sekä tilan muuttaminen että pyyntöjen lähettäminen palvelimelle ovat sivuvaikutuksia. Toinen ongelma on tapahtumien käsittely, joka sekin perustuu täysin sivuvaikutuksiin. Molemmat näistä ongelmista voidaan kuitenkin ratkaista yhdistämällä funktionaalinen ohjelmointi reaktiiviseen ohjelmointiin.

3.4. Funktionaalinen reaktiivinen ohjelmointi

Funktionaalinen reaktiivinen ohjelmointi (*Functional Reactive Programming, FRP*) yhdistää keskenään funktionaalisen ohjelmoinnin ja reaktiivisen ohjelmoinnin. [24] Sen lähtökohdiana on tapahtumien kuvaaminen funktionaalisen tietorakenteena, jota kutsutaan tapahtumavirraksi[3] (*stream*). [25] Tapahtumavirtaa voi ajatella laiskana listana tapahtumia. Virran pituus ei ole etukäteen tiedossa, vaan tapahtumia voi ohjelman suorituksen aikana olla mielivaltaisen määrä kussakin tapahtumavirrassa. Esimerkki tapahtumavirrasta voisi olla tietyn käyttöliittymässä olevan napin painallukset: virtaan lisätään uusi tapahtuma aina, kun käyttäjä painaa kyseistä nappia. [26]

Koska tapahtumavirta on funktionaalinen tietorakenne, sitä voi myös käsitellä funktionaalisilla työkaluilla. Tapahtumavirrasta voi luoda uuden virran muuttamalla sen tapahtumia `map`-funktioilla tai suodattamalla sen tapahtumia `filter`-funktioilla. Tapahtumavirtoja voi myös yhdistää keskenään, ja uuden tapahtumavirran luominen ei muuta alkuperäistä virtaa, vaan alkuperäinenkin virta säilyy edelleen käytettävissä. Tässäkin mielessä tapahtumavirrat ovat kuin mikä tahansa muuttumaton

¹Tässä rinnakkaisuudella tarkoitetaan englanninkielistä termiä *concurrency*, eli yhtä aikaa suoritettavien säikeiden välistä kommunikaatiota ja yhteisten resurssien käyttöä. Valitettavasti myös termi *parallelism* käännetään rinnakkaisuudeksi. Tämä on omiaan aiheuttamaan sekaannuksia näiden käsitteiden välille.

lista. Funktiot, joilla tapahtumavirtoja käsitellään, ovat puhtaita. Vaikka tapahtumien syntyminen ja tapahtumavirrasta poistuminen perustuu luonnostaan sivuvaikutuksiin, kaikki tapahtumavirtojen käsittely on puhtaan funktionaalista. [26]

Tapahtumavirtoja voidaan myös käsitellä. Tapahtumavirran käsittelijä on funktio, jota kutsutaan jokaista virran tapahtumaa kohden. Koska tällaisen funktion paluuarvoa ei tallenneta mihinkään, perustuu sen toiminta sivuvaikutuksiin. Niinpä käsittelijän toiminta liittyykin yleensä käyttäjän kanssa vuorovaikutukseen, esimerkiksi käyttöliittymän päivitykseen. [26]

Toinen funktionaaliselle reaktiiviselle ohjelmoinnille keskeinen käsite on signaali[3] (*signal*). Signaali on arvo, joka vaihtelee ajan myötä. [25] Signaalin arvo voi perustua toisiin signaaleihin tai tapahtumavirtoihin. Signaaleilla kuvataankin nykyhetkellä voimassaolevaa tilaa, siinä missä tapahtumavirroilla kuvataan tilan muutoksia. Signaalilla on jokaisella ajanhetkellä tietty voimassaoleva arvo, toisin kuin tapahtumavirroilla. Kuten tapahtumavirtoja, signaalejakin käsitellään täysin funktionaalisilla menetelmillä. Signaali voidaan esimerkiksi muodostaa toisesta signaalista suorittamalla sen arvolle jokin funktio, samoin kuin `map`-funktiolla muodostetaan tapahtumavirrasta uusi tapahtumavirta.

Signaaleista voidaan myös vastaavasti muodostaa tapahtumavirtoja, joissa jokainen signaalin arvon muutos synnyttää virtaan uuden tapahtuman. Signaalit ja tapahtumavirrat ovat hyvin läheisesti toisiinsa liittyvät konseptit, ja niiden tärkein ero on se, että signaalilla on jatkuvasti määritelty arvo, mutta tapahtumavirroilla ei ole yksittäisten tapahtumien välissä mitään määriteltyä arvoa. [26]

Funktionaalinen reaktiivinen ohjelmointi on myös deklaratiiivista. Tapahtumavirtojen ja signaalien suhteet toisiinsa kuvaavat, *mitä* halutaan saavuttaa, ei *miten* se saavutetaan. [26]

Reaktiivinen funktionaalinen ohjelmointi on web-ohjelmoinnissa hyödyllisimmillään käyttöliittymässä, missä ohjelman toiminta perustuu hyvin suurilta osin käyttäjän toimenpiteisiin. Niinpä funktionaalista reaktiivista ohjelmointia toteuttavia kirjastoja on tehty erityisesti selainohjelmointiin. Kirjastoja on tehty eri kielille ja eri laajuuksina. Yleisiä kirjastoja ovat esimerkiksi JavaScriptille tehty Bacon.js[27], joka tarjoaa käyttäjälle ainoastaan tapahtumavirrat ja signaalit, jolloin käyttöliit-

tymän toteutus jää ohjelmoijan tai toisen kirjaston vastuulle, sekä ClojureScriptille tehty re-frame[28], joka tarjoaa tapahtumavirtojen ja signaalien lisäksi myös signaaleina mallinnetut käyttöliittymäkomponentit.

Palvelinohjelman puolella reaktiivinen funktionaalinen ohjelmointi ei ole yhtä käytännöllinen, kun palvelimen toiminta perustuu HTTP-pyyntöihin ja niihin vastaamiseen. REST-periaatteen mukainen palvelin ei säilytä käyttäjäkohtaista tilaa, vaan jokainen HTTP-pyyntö sisältää kaiken pyynnön palvelemiseen tarvittavan tiedon. Tämän takia signaalit eivät ole yleensä hyödyllinen käsite web-palvelimelle, sillä signaalit liittyvät oleellisesti ohjelman tilaan. Sen sijaan jokainen HTTP-pyyntö on käytännössä vain yksi funktiokutsu, ja ainoa muodostuva tapahtumavirta on palvelimelle saapuvat pyynnöt. Siksi funktionaalisesta reaktiivisesta ohjelmoinnista ei saada yhtä suurta hyötyä kuin käyttöliittymäohjelmoinnissa, vaikka palvelimen toiminta onkin pohjimmiltaan reaktiivista.

4. TUTKIMUS

Tutkimuksen tavoitteena oli selvittää, miten funktionaaliset ohjelmointikielet soveltuvat web-ohjelmistokehitykseen. Erityisesti pyrittiin löytämään eroavaisuuksia funktionaalisten ja imperatiivisten kielten käytön välillä. Tässä luvussa esitetään tehdyn tutkimuksen lähtökohdat ja menetelmät sekä tutkimuksen tulokset.

4.1. Tutkimusmenetelmä

Tutkimukseen kuului sekä haastattelututkimus että kyselytutkimus. Molemmissa osallistujina oli Solita Oy:n työntekijöitä, jotka olivat osallistuneet funktionaalisella ohjelmointikielellä tehtyyn web-ohjelmistoprojektiin. Vain haastattelututkimus tehtiin osana tätä diplomityötä. Kyselytutkimus oli tehty jo aikaisemmin, ja sen tulokset saatiin käyttöön tätä työtä varten.

4.1.1. Haastattelututkimus

Varsinainen tutkimus suoritettiin teemahaastattelututkimuksena. Teemahaastattelulla tarkoitetaan haastattelua, jossa edetään ennalta valmisteltujen avointen kysymysten ja teemojen pohjalta. Valmiita vastausvaihtoehtoja ei ole, vaan vastaukset rakentuvat täysin haastateltavien oman kokemuksen pohjalta. Kysymysten ja teemojen valinnalla ohjataan haastattelu haluttuun suuntaan ja aiheisiin. Teemahaastattelulla kerättävä aineisto on aina tyypiltään laadullista. Tutkimuksen tekijän täytyy siis pystyä löytämään haastattelumateriaalista oleelliset tulokset ja niiden tulkinnat. [29]

Haastattelussa käytetyt teemat olivat seuraavat:

1. Projektissa käytetty funktionaalinen kieli tai kielet
2. Tekijän kokemus funktionaalisista ja imperatiivisista kielistä ennen projektia

3. Funktionaalisen kielen opettelun haasteet
4. Funktionaalisen ohjelmoinnin edut verrattuna imperatiivisiin kieliin
5. Imperatiivisen ohjelmoinnin edut verrattuna funktionaalisiin kieliin
6. Tuottavuus funktionaalaisella kielellä verrattuna imperatiivisiin kieliin
7. Frameworkien ja kirjastojen saatavuus ja erot verrattuna Javan tai C#:n frameworkkeihin ja kirjastoihin
8. Eri teknologioilla tehtyjen projektien ylläpidon haasteet
9. Eroavuudet eri funktionaalisten kielten välillä
10. Yleinen mielipide funktionaalaisella kielellä tehdyistä projekteista
11. Teknologiaavalintojen vaikutus tiimityöskentelyyn

Jo ennalta oli tiedossa, että ohjelmoinnin tuottavuutta eri kielillä on vaikea arvioida tai verrata. Teema haluttiin kuitenkin mukaan haastattelututkimukseen, sillä sitä oli kysytty myös työn osana olevassa kyselytutkimuksessa.

4.1.2. Kyselytutkimus

Teemahaastattelujen lisäksi tutkimusta varten saatiin tulokset aiemmin Solitalla tehdystä kyselytutkimuksesta. Tutkimuksessa kerättiin erään projektin henkilökunnan kokemuksia funktionaalisen kielen käytöstä 6 ja 12 kuukautta projektin aloittamisen jälkeen. Kummallakin ajanhetkellä projektin henkilökunnalta kysyttiin samat kysymykset, ja näin voitiin seurata tapahtunutta kehitystä. Tutkimukseen osallistui 8 projektitiimin jäsentä, joista yksi poistui projektista kyselyiden välisenä aikana. Projektissa käytetty funktionaalinen ohjelmointikieli oli Clojure.

Kyselytutkimuksessa kysyttiin seuraavia asioita:

1. Kokemus vuosina eri ohjelmointikielistä ja menetelmistä
2. Arvio omista taidoista asteikolla 1-5 eri ohjelmointikielistä ja menetelmistä

3. Arvio omasta tuottavuudesta ja ohjelmakoodin rivimäärästä, molemmissa tapauksissa Clojurella Javaan verrattuna
4. Vapaita kommentteja projektin eri osa-alueista

Jälkimmäisessä kyselyssä kysyttiin lisäksi tekijöiden mielipidettä tulevien projektien teknologiavalinnoista.

4.1.3. Tutkimukseen osallistujat

Haastateltaviin kuului sekä ohjelmistosuunnittelijoita, ohjelmistoarkkitehtejä että projektipäälliköitä, jotka olivat työskennelleet sekä projekteissa, missä pääosa kehityksestä tehtiin funktionaalisella kielellä, että projekteissa, missä pääasiallinen ohjelmointikieli oli imperatiivinen. Kaikilla haastatelluilla työprojektissa käytetty funktionaalinen kieli oli Clojure.

Tutkimukseen osallistui yhteensä 10 Solitan työntekijää, joista 7 oli ohjelmistosuunnittelijoita, 2 ohjelmistoarkkitehtejä ja 1 projektipäällikkö. Kategorioiden rajat eivät Solitan projekteissa ole tarkkoja, vaan myös arkkitehdit ja joissain tapauksissa myös projektipäälliköt osallistuvat varsinaiseen ohjelmointityöhön.

Kaikki haastateltavat olivat osallistuneet projektiin, jossa käytettiin Clojurea. Haastateltavista kaksi oli käyttänyt Scalaa, yksi F#ia ja yksi Haskellia. Myös JavaScript mainittiin useamman kerran, koska se sisältää joitain funktionaalisia ominaisuuksia, vaikka onkin lähtökohtaisesti imperatiivinen kieli. Erityisesti JavaScriptin uusin variantti EcmaScript 6 sisältää huomattavan määrän funktionaalisten ohjelmointikielten ominaisuuksia.

Suurin osa oli saanut ensikosketuksensa funktionaaliseen ohjelmointiin meneillään olevassa projektissa. Kolmella haastateltavalla oli pidempi Clojure-kokemus. He olivat toimineet projekteissa pääarkkitehteinä ja mentoreina, ja olivat osaltaan myös vaikuttaneet siihen että Clojure ylipäätään otettiin projekteissa käyttöön. Haastateluissa esiintynyt F#- sekä Haskell-kokemus oli peräisin haastateltavien vapaa-ajan harrastuneisuudesta.

Haastateltavien aikaisempi kokemus oli lähes kaikilla Java-ohjelmoinnista. Yksi haastateltavista oli siirtynyt suoraan C++/Symbian-ohjelmoinnista Clojureen.

Kaikki haastateltavat olivat siis siirtyneet funktionaaliseen ohjelmointiin jonkin imperatiivisen kielen parista.

Kyselytutkimukseen osallistuneilla oli vuoden projektityöskentelyn jälkeen keskimäärin 1,4 vuoden kokemus Clojure-ohjelmoinnista ja 8,0 vuoden kokemus Java-ohjelmoinnista. Puolella osallistujista Clojure-kokemus oli suureksi osaksi vapaa-ajalla tai aikaisemmissa projekteissa kerättyä, puolella kokemusta oli ainoastaan meneillään olevasta projektista.

Koska Clojure ja Java olivat kaikille haastatelluille tuttuja, keskitytään tässä tutkimuksessa pääosin vertailemaan niitä. Koska Clojure on dynaamisesti tyyplitetty kieli, vertaillaan Javaa staattisen tyyppityksen osalta Scalaan ja Haskellisiin.

4.1.4. Tutkimuksen luotettavuus

Tutkimuksen tuloksia arvioitaessa on myös arvioitava tutkimuksen luotettavuutta. Tutkimuksen luotettavuuteen vaikuttavat tekijät riippuvat suuresti tutkimusmenetelmästä. Tässä tutkimuksessa tutkimusmenetelmänä käytettiin teemahaastattelua. Teemahaastattelun luotettavuuteen vaikuttavia asioita ovat teemojen valinta, haastattelun tulosten tallennus ja analysointi sekä haastateltavien valinta. [30]

Teemojen valinta pyrittiin tekemään siten, että teemat eivät sisällä ennakkoletuksia lopputuloksesta, vaan vertailtavat asiat pyrittiin esittämään mahdollisimman yhdenvertaisina. Teemoissa pyydettiin haastateltavaa kertomaan yleisesti kahden eri asian eroista, tai ensin asian A eduista B:hen nähden, sitten asian B eduista A:han nähden. Teemat pyrittiin myös kuvaamaan siten, että kaikki haastateltavat ymmärtäisivät teemat samalla tavalla eikä väärinymmärryksiä syntyisi.

Kaikki haastattelut äänitettiin ja äänitykset purettiin tekstiksi haastattelujen jälkeen. Näin vältettiin mahdollisuus, että haastattelutilanteessa jokin asia olisi jäänyt kirjaamatta. Äänitysten purkamisessa pyrittiin mahdollisimman kattavasti kirjaamaan kaikki haastattelussa esiintulleet asiat. Kaikilta haastateltavilta pyydettiin lupa haastattelun äänittämiseen.

Haastateltaviksi henkilöiksi valittiin Solita Oy:n työntekijöitä, joilla on työkokemusta sekä imperatiivisilla että funktionaalisilla kielillä tehdyistä projekteista. Haastatteluun otettiin eri rooleissa työskennelleitä henkilöitä: ohjelmoijia, ohjelmis-

toarkkitehtejä sekä projektipäälliköitä. Kaikki haastateltavat valittiin Solita Oy:llä funktionaalaisella kielellä tehdyistä projekteista, jotta kaikilla haastateltavilla varmasti olisi kokemusta funktionaalisesta ohjelmoinnista. Projekteja, joista haastateltavia valittiin, oli yhteensä kolme kappaletta. On huomioitava, että kapea projektivalikoima voi aiheuttaa tulosten vääristymistä, etenkin kun kyseiset projektit ovat ensimmäisten Solita Oy:ssä funktionaalisilla kielillä tehtyjen projektien joukossa. Tämän vuoksi haastateltavia pyydettiin keskittymään nimenomaan ohjelmointikieliin ja -paradigmoihin, ettei yksittäisen projektin mahdolliset ongelmat heijastuisi tuloksiin liikaa.

Tutkimuksen osana olleeseen kyselytutkimukseen ei ollut vaikutusmahdollisuuksia, joten sen luotettavuutta on arvioitava tietämättä tarkalleen, miten tutkimus on tehty. Käytössä oli ainoastaan valmiiksi tehtyt yhteenvedot tuloksista, ei raakaa tutkimusdataa. Kahden kyselyn kysymyksistä ainakin osa on sellaisia, joiden tulosten kehittyminen on helposti ennakoitavissa. Sekä ohjelmoijien arvio omista taidoista että tuottavuudesta on kasvanut odotusten mukaisesti kyselyiden välillä. Myös kokemuksen pituus vastaa kyselyajankohtia. Näiden tietojen perusteella kyselytutkimuksen tuloksia voidaan pitää riittävän luotettavina.

4.1.5. Tulosten käsittely ja esitys

Haastattelujen tulokset käsiteltiin käymällä läpi tehdyt haastattelut ja erittelemällä haastateltavien tekemät havainnot teemoittain. Näin saatiin temakohtaisesti lista tehdyistä havainnoista ja siitä, miten moni haastateltava teki kunkin havainnon.

Aiemmin tehdystä kyselytutkimuksesta saatiin käyttöön valmiiksi tehtyt taulukot kunkin osallistujan vastauksista kysymyksiin sekä 6 että 12 kuukauden projektityöskentelyn jälkeen.

Tutkimuksen tulokset on jaettu lukuihin, joista jokaisessa esitellään yksi aihepiiri, jossa funktionaalinen ohjelmointi eroaa imperatiivisesta ohjelmoinnista. Aihepiirit on kerätty sekä haastattelututkimuksen että kyselytutkimuksen tuloksista.

4.2. Funktionaalisten ohjelmointikielten ominaisuudet

Haastatteluissa useimmin ilmi tulleet asiat liittyivät funktionaalisten ohjelmointikielten ominaisuuksiin ja niiden hyötyihin ja haittoihin web-ohjelmistokehityksessä. Nämä asiat voidaan jakaa kahteen kategoriaan: datan käsittelyyn liittyvät ominaisuudet sekä funktioiden puhtaus ja siitä seuraavat hyödyt.

4.2.1. Datan käsittely

Nykyaikaisen web-ohjelmiston palvelinohjelma on usein vain liukuhihna, joka syöttää dataa tietokannasta käyttöliittymälle ja päinvastoin. Siksi ohjelmointikieli, joka sisältää datan käsittelyä helpottavia ominaisuuksia, tarjoaa myös paremman lähtökohdan web-ohjelmointiin. Haastatteluissa mainittiin tähän liittyen sekä Clojuren tietorakenteet että funktionaaliset työkalut datan käsittelyyn.

Olio-ohjelmoinnissa lähtökohtana on se, että jokaista tietoalkiota vastaa tietyn luokan olio, joka on vastuussa alkion datasta sekä siihen liittyvistä operaatioista. Tällainen olio, joka on tarkoitettu ainoastaan datan kuljettamiseen tunnetaan termillä *data transfer object (DTO)*, tiedonsiirto-olio. Se sisältää ainoastaan dataa sekä mahdollisuuden sen lukemiseen ja muokkaamiseen. [31] Esimerkki henkilön kuvaamisesta tiedonsiirto-oliona on esitetty ohjelmassa 4.1.

Olio-ohjelmoinnin lähestymistapa on kuitenkin tarpeettoman raskas liukuhihnallisiin, jossa dataan ei välttämättä liity mitään operaatioita, ja jossa alkion sisältämän datan määrä ja laatu voi muuttua matkan varrella. Clojure tarjoaakin tähän ongelmaan vaihtoehtoisen ratkaisun.

Clojuressa tiedonsiirto-olion korvaa assosiatiivinen taulukko (*map*), jonka avaimet ja niitä vastaavat arvot voivat olla mitä tahansa tyyppiä ja niitä voi olla mielivaltainen määrä. Yleisesti avaimina käytetään kuitenkin avainsanoja (*keyword*). Avainsana on Clojureen kuuluva tietotyyppi, joka on suunniteltu nimenomaan assosiatiivisen taulukon avaimeksi. Ohjelmassa 4.2 on esimerkki henkilöä kuvaavasta taulukosta. Taulukko koostuu avain–arvo-pareista, joissa avain on aina kaksoispisteellä alkava avainsana.

Ohjelma 4.1: Esimerkki henkilön kuvaamisesta Javalla tiedonsiirtoluokan avulla

```
1 public class Person {
2     private String name;
3     private int age;
4     private String email;
5
6     public Person(String name, int age, String email) {
7         this.name = name;
8         this.age = age;
9         this.email = email;
10    }
11
12    public String getName() {
13        return name;
14    }
15    public void setName(String name) {
16        this.name = name;
17    }
18
19    public int getAge() {
20        return age;
21    }
22    public void setAge(int age) {
23        this.age = age;
24    }
25
26    public String getEmail() {
27        return email;
28    }
29    public void setEmail(String email) {
30        this.email = email;
31    }
32 }
33
34 Person p = new Person("John Smith", 28, "john.smith@gmail.com");
```

Ohjelma 4.2: Esimerkki henkilön kuvaamisesta Clojurella assosiatiiivisena taulukkona

```

1 (def person {:name "John Smith"
2              :age 28
3              :email "john.smith@gmail.com"})

```

Ohjelma 4.3: Esimerkki henkilön kuvaamisesta Scalalla case classina

```

1 case class Person(name: String, age: Int, email: String)
2 val person = Person("John Smith", 28, "john.smith@gmail.com")

```

Assosiatiiivisella taulukolla ei ole staattista rakennetta, vaan sen sisältämät avain-arvo-parit voidaan määritellä vasta ohjelman ajon aikana dynaamisesti. Niinpä ohjelmoijan ei tarvitse tehdä jokaiselle mahdolliselle tietomallille omaa luokkaa. Eräessä haastattelussa mainittiin esimerkkinä tilanne, missä tiettyä dataa käytetään kahdessa eri muodossa: luettelossa, jossa tarvitaan ainoastaan alkion nimi ja ID; sekä alkion tarkempien tietojen tarkastelussa, missä siitä tarvitaan kaikki tiedot. Jos käytössä olisi staattisesti tyyppitetty tiedonsiirto-olio, täytyisi valita jokin huono ratkaisu: kahden eri luokan tekeminen eri käyttötarkoituksiin, tai tarpeettomien kenttien jättäminen tyhjäksi jos niitä ei tarvita, tai tarpeettoman datan siirtäminen kun turhatkin kentät on täytetty. Clojure-taulukko puolestaan voi sisältää ainoastaan tarpeelliset kentät tilanteen mukaan.

Clojuren mallikaan ei ole täysin ongelmaton. Sen suurimpana riskinä haastatelmissa pidettiin ajonaikaisten virheiden mahdollisuutta, kun tietomallin sisältämiä kenttiä ei tarkasteta missään vaiheessa. Staattisesti tyyppitetystä tiedonsiirto-oliosta saa virheilmoituksen jo käännoaikana, jos ohjelmoija yrittää käyttää tietokenttää, jota ei ole olemassa. Riskinä nähtiin myös se, että jos ohjelmalla on ulkoisia rajapintoja, on helppoa päästää vahingossa vääränlaista dataa rajapinnan kutsujalle. Mahdollista on vahingossa niin tarpeettoman datan lähettäminen (mikä kuluttaa tarpeettomasti kaistaa ja hidastaa rajapinnan toimintaa), vääräntyyppisen datan lähettäminen (mikä aiheuttaa virhetilanteita rajapinnan käyttäjälle) kuin arkaluontaisen datan paljastaminen (mikä voi olla vakavakin tietoturvariski).

Clojuren versiossa 1.9 on lisätty eräs ratkaisu tähän ongelmaan. *Clojure Spec* on tapa määritellä, mitä kenttiä tietoalkio saa sisältää, ja mitä tyyppiä niiden tulee olla.

Ohjelmoija pystyy itse valitsemaan, missä vaiheessa ohjelman suoritusta nämä määritelmät tarkastetaan. Useimmat haastateltavat kertoivat, että he ovat käyttäneet vastaavaa kirjastoa tekemissään projekteissa. Erityisen hyödylliseksi se koettiin automaattisessa testauksessa ja rajapintojen määrittelyssä. Koska dynaamisen tyyppityksen vuoksi määritykset voidaan tarkastaa ainoastaan ohjelmaa ajettaessa, suurin hyöty saadaan automaattitestauksen yhteydessä, kun testit epäonnistuvat, jos data ei vastaa määrittelyä. Tuotantokäytössä haastateltavat olivat yleensä jättäneet varmistukset pois, sillä saavutettu hyöty on huomattavasti pienempi ja tehokkuushaitta todellinen varsinkin suurten datamäärien kohdalla. Hyödyn saaminen vaatii tietysti siinä tapauksessa kaikkien rajapintojen automaattitestaamista, sillä muutoin mikä tahansa edellämmainituista riskeistä voi toteutua. Eräät haastateltavista olivat tehneet avoimen lähdekoodin kirjaston, joka yhdistää REST-rajapinnan määrittelyyn, datan validoinnin sekä rajapintadokumentaation generoinnin [32]. Kirjaston avulla voidaan luoda uudelleenkäytettäviä sekä koostettavia tietotyyppimäärittelyjä ja muodostaa niistä yhdellä kertaa sekä rajapintadokumentaatio että rajapinnan kautta kulkevan datan validaatio.

Toisaalta myös staattisesti tyyppitetyissä funktionaalisissa kielissä on ominaisuuksia, jotka helpottavat datan käsittelyä Javaan verrattuna. Scalan ominaisuuksista mainittiin *case classit*, jotka ovat ikään kuin yksinkertaistettuja tiedonsiirtoluokkia: niille ei tarvitse määritellä erikseen rakentajaa tai metodeja datan hakemiseen ja asettamiseen, ne ovat lähtökohtaisesti muuttumattomia ja niitä voi käyttää yhdessä Scalan hahmontunnistuksen (*pattern matching*) kanssa. Ohjelmassa 4.3 on esimerkki henkilön esittämisestä *case classin* avulla. Suurin osa staattisen tyyppityksen tuomista haitoista on kuitenkin olemassa myös Scalassa.

Funktionaalinen tapa käsitellä dataa sopii myös erityisen hyvin liukuhihnalliin. Datan rakennetta muokkaavia funktioita on mahdollista yhdistää. Niillä pystyy muokkaamaan isompaa datajoukkoa `map`-funktion avulla. Datajoukkoa voi suodattaa `filter`-funktioilla ja koostaa `reduce`-funktioilla. Funktionaalisilla työkaluilla voidaan siis pienistä, monikäyttöisistä palasista koota helposti ymmärrettävä ja muunneltava liukuhihna, jolla tieto saadaan siirrettyä tietokannasta käyttöliitty-

mälle ja takaisin kulloiseenkin tarpeeseen sopivassa muodossa. Tämä liittyy edelleen funktionaalisen kielen ilmaisuvoimaan ja deklaratiivisuuteen.

4.2.2. Puhtaus

Funktioiden puhtaus mainittiin haastatteluissa kahdessa eri yhteydessä: ohjelman testattavuus ja yksinkertaisuus. Nämä molemmat liittyvät myös muihin kategorioihin, kuten ilmaisuvoimaan ja tuottavuuteen.

Puhtaan funktion paluuarvo riippuu ainoastaan sen syötteistä. Samoilla syötteillä paluuarvo on aina sama, ja tämä helpottaa funktion testaamista huomattavasti. Hyvän testin kirjoittamiseksi ohjelmoijan tarvitsee vain määritellä funktion syötteet sekä odotettu paluuarvo. Koska funktiolla ei ole sivuvaikutuksia, ei testitapauksen tarvitse ottaa kantaa mihinkään funktion ulkopuoliseen tilaan tai riippuvuuksiin, koska sellaisia ei ole. Testien kirjoittamisen helpous kannustaa myös ohjelmoijaa kirjoittamaan testejä, mikä tekee ohjelmasta luotettavamman ja virheiden havaitsemisesta helpompaa. Haastatteluissa mainittiin funktionaalisen ohjelmoinnin siirtävän testauksen painopistettä yksikkötesteihin eli yksittäisten funktioiden testaukseen. Yksikkötestien ajaminen on huomattavasti nopeampaa kuin laajempien, koko ohjelman toiminnallisuutta tutkivien testien ajaminen, sillä ne eivät vaadi minkäänlaista alustusta. Tällöin ohjelmoija myös ajaa testit mieluummin, kun niiden valmistumista ei tarvitse odotella. Tämäkin kasvattaa ohjelman luotettavuutta. Testien sanottiin myös auttavan ohjelmoijaa ymmärtämään ohjelman toiminta, kun funktion toiminnallisuuden voi selvittää myös lukemalla sille kirjoitetut testit, varsinkin jos kyseessä on hieman monimutkaisempi funktio, jonka toiminnallisuus ei käy ilmi yhdellä silmäyksellä ohjelmakoodista. Näin tiimityöskentely helpottuu, kun muiden kirjoittaman koodin ymmärtäminen on yksinkertaisempaa.

Ohjelman yksinkertaisuuteen puhtauden kerrottiin vaikuttavan siten, että funktioita kutsuessaan ohjelmoijan ei tarvitse miettiä, onko kutsuminen turvallista. Koska puhtaalla funktiolla ei ole sivuvaikutuksia, ohjelmoija voi luottaa siihen, ettei kutsuminen aiheuta ongelmia. Tämä helpottaa erityisesti laajojen ohjelmistojen kehittämistä, kun ohjelmoijan ei tarvitse välttämättä ottaa koko ohjelman toiminnasta selvää. Vaikkei asia välttämättä parantaisi koko ohjelman yksinkertaisuutta, on

yhden ohjelmoijan kerrallaan käsittelemä osa ohjelmasta huomattavasti yksinkertaisempi.

Puhtaus parantaa myös koodin laatua. Sellaista tilannetta ei pysty vahingossa syntymään, missä ohjelmoijalta huomaamatta jäänyt sivuvaikutus muuttaa ohjelman toimintaa muualla kuin sillä hetkellä suoritettavassa ohjelmakoodissa. Laajalle ulottuvien sivuvaikutusten aiheuttamia virheitä ei välttämättä löydetä automaattitesteilläkään, koska usein yksittäinen testi testaa vain ohjelman yhden osan toimintaa, jolloin vaikutukset ohjelman toiseen osaan jäävät paljastumatta.

4.3. Funktionaalisen ohjelmoinnin tehokkuus

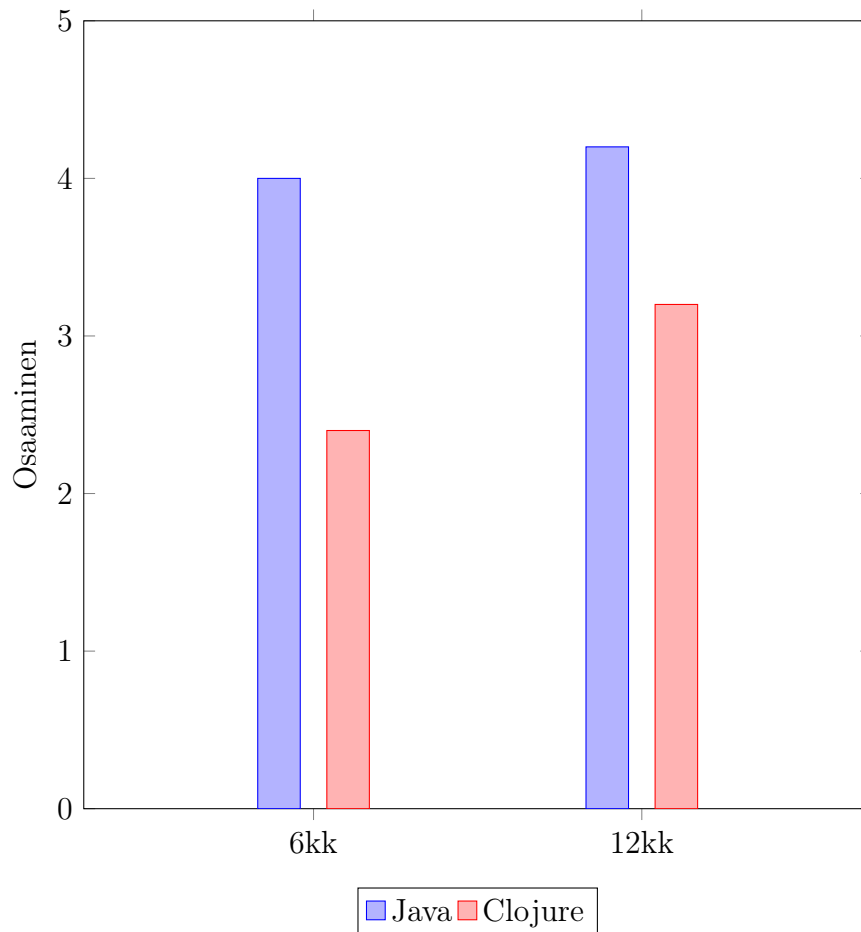
Funktionaalaisella kielellä tehdyn ohjelmointityön tehokkuutta selvitettiin kysymällä haastateltavien ja kyselytutkimuksen vastaajien näkemystä omasta tuottavuudesta funktionaalaisella ohjelmointikielellä. Lisäksi haastatteluissa tuli esiin ohjelmointikielen ilmaisuvoima tehokkuuteen vaikuttavana asiana.

4.3.1. Tuottavuus

Kuten ennalta odotettiin, monet haastateltavista totesivat, että eri kielillä ohjelmoinnin tuottavuutta on vaikea vertailla keskenään. Erityisesti kokeneemmat Clojure-ohjelmoijat toivat esiin näkemyksen, että käytetty kieli vaikuttaa tuottavuuteen eniten ohjelmoijan motivaation kautta. Jos kieli on ohjelmoijan mielestä miellyttävä käyttää, vaikuttaa se tuottavuutta parantavasti. Suurin osa haastateltavista arvioi pystyvänsä Clojurella vähintään yhtä hyvään tuottavuuteen kuin Javalla, vaikka täsmällistä arviota ei kysyttykään.

Clojure olikin suurimmalle osalle tutkimuksen osallistujista huomattavasti miellyttävämpi kieli kuin Java. Kyselytutkimukseen vastanneista kuusi kahdeksasta käyttäisi mieluummin Clojurea kuin Javaa, jos saisi itse valita projektissa käytettävän kielen. Scala nähtiin Javan kanssa melko tasavertaisena vaihtoehtona.

Haastateltavat, joilla oli vähemmän Clojure-kokemusta, kiinnittivät huomiota uuden kielen opetteluun. Vallitseva näkemys oli, että sekä funktionaalinen paradigma että erityisesti Clojure Lisp-murteena vaativat paljonkin opettelemista. Kaikki tutkimukseen osallistuneet olivat siirtyneet funktionaalisen ohjelmoinnin pariin impera-

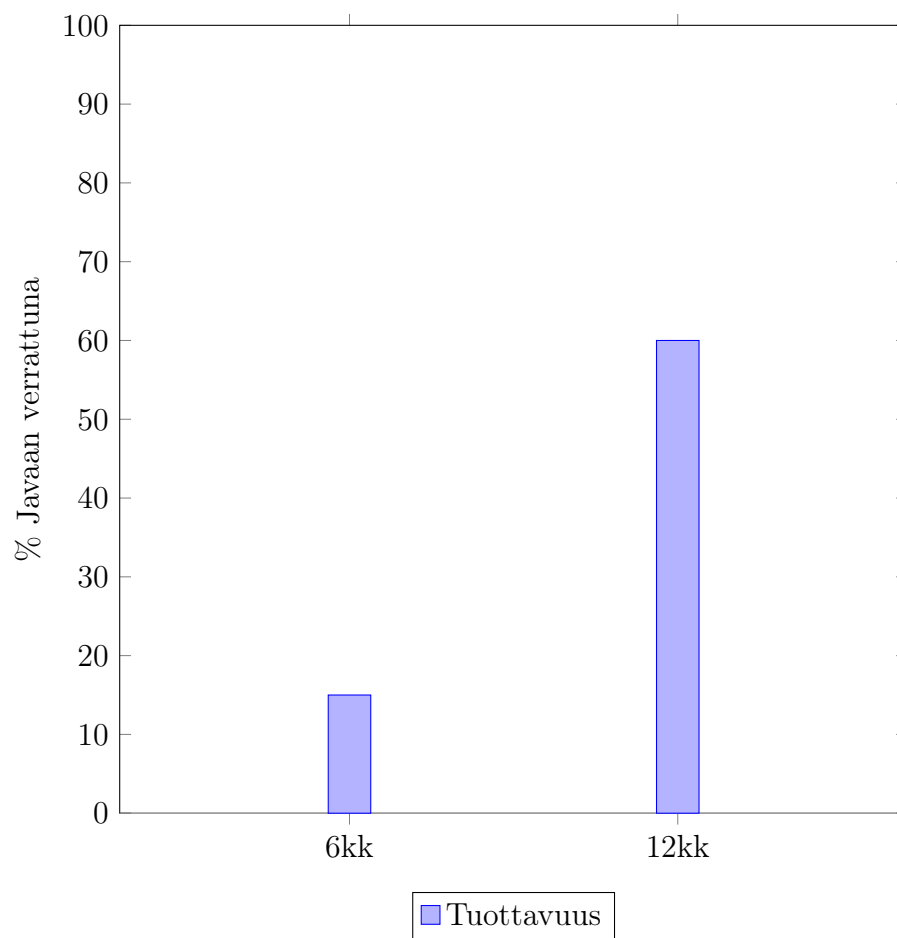


Kuva 4.1: Kyselytutkimukseen osallistuneiden arvio Java- ja Clojure-taidoista asteikolla 1-5

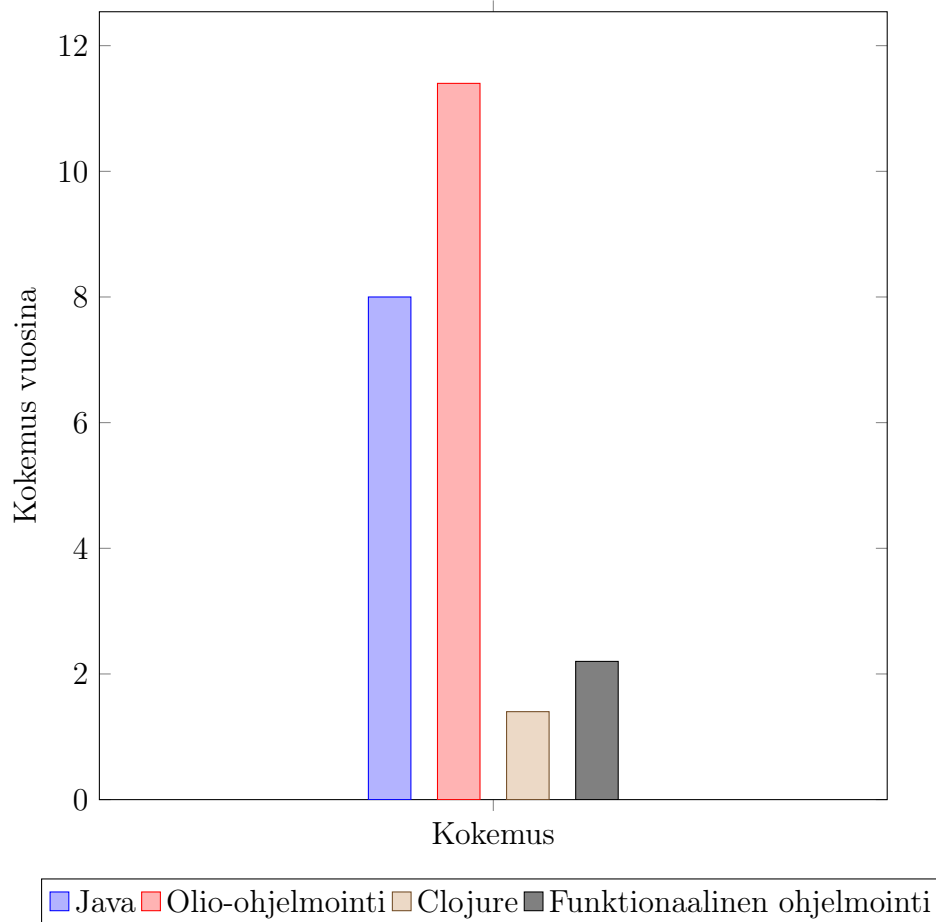
tiivisesta ohjelmoinnista, ja paradigman vaihtaminen mainittiin useimmin opetteluhaasteena; useiden vuosien aikana muodostuneista ajattelutavoista ja tottumuksista ei ole helppo päästä eroon. Paradigman lisäksi suurimmiksi ongelmiksi Clojuren oppimisessa mainittiin harvoin tarvittut, edistyneet ominaisuudet kuten makrot ja transducerit. Lisp-syntaksi aiheutti päänvaivaa lähinnä opettelu alkuvaiheessa.

Kyselytutkimuksessa kysyttiin projektihenkilöstön näkemystä taidoistaan sekä Javan että Clojuren osalta. Kuvassa 4.1 on esitetty kysymyksen tulokset vastausten keskiarvoina. Vastauksista käy ilmi, että Javasta Clojureen siirtyminen ei ole nopea prosessi. Vielä vuoden jälkeenkin vastaajat arvioivat Clojure-taitonsa huomattavasti heikommiksi kuin Java-taitonsa. Selkeä kehittyminen tuloksista kuitenkin näkyy.

Pelkkä omien taitojen arviointi huonommiksi Clojurella kuin Javalla ei kuitenkaan tarkoita sitä, että Clojure olisi tuottavuuden kannalta huonompi valinta. Tutkimuksessa kysyttiin myös henkilöstön arviota omasta tuottavuudestaan Clojurella Javaan



Kuva 4.2: Kyselytutkimukseen osallistuneiden arvio omasta tuottavuudesta Clojurella Javaan verrattuna



Kuva 4.3: Kyselytutkimukseen osallistuneiden kokemus vuoden projektityöskentelyn jälkeen Javan ja Clojuren käytöstä sekä funktionaalisesta ja olio-ohjelmoinnista vuosina

verrattuna. Tulokset on esitetty kuvassa 4.2. Jo puolen vuoden jälkeen tekijät arvioivat olevansa hieman tuottavampia Clojurella kuin Javalla. Samaan aikaan omat Clojure-taidot arvioitiin vielä varsin heikoiksi. Vuoden jälkeen arvio tuottavuudesta oli noussut vielä suuremmaksi: tekijät arvioivat tuottavuutensa keskimäärin 60 % paremmiksi kuin Javalla.

Kyseisen projektin alussa suurimmalla osalla tekijöistä ei kuitenkaan ollut lainkaan Clojure-kokemusta. Kuten kyselytutkimuksen tuloksista näkyy, vuoden projektityöskentelyn jälkeen keskimääräinen Clojure-kokemus oli vain n. 16 kuukautta. Vertailun vuoksi tekijöiden keskimääräinen kokemus Java-ohjelmoinnista oli jopa 8 vuotta. Olio-ohjelmoinnista kokemusta oli keskimäärin yli 11 vuotta, funktionaalisesta ohjelmoinnista vain noin 2 vuotta. Tulokset kokemuksen osalta on esitetty kuvassa 4.3. Jos näin huomattavasta kokemuserosta huolimatta tekijät arvioivat olevansa Clojurella merkittävästi tuottavampia kuin Javalla, voidaan päätellä, että osaami-

nen nousee kohtuullisessa ajassa täysin riittävälle tasolle päivittäistä työskentelyä ajatellen.

Taitavaa mentoria pidettiin erittäin oleellisena sekä kokemattomien ohjelmoijien tuottavuudelle että ylipäänsä projektin onnistumiselle, jos projektin henkilöstöstä suurin osa on kokemattomia Clojure-ohjelmoijia. Saman tosin todettiin olevan tosi missä tahansa projektissa, mutta Java-projekteissa erityisen mentorin tarve on usein vähäinen, sillä osaaminen on keskimäärin parempaa. Java-projekteissa mentorin tarve mainittiin lähinnä yksittäisille kokemattomille henkilöille, ei koko projektitiimille. Kahdeksan hengen projektitiimissä ollutta kahta mentoria pidettiin riittävänä määränä. Vain yhtä mentoria pidettiin liian suurena riskinä, jos suurin osa tiimiläisistä on kokemattomia funktionaalisessa ohjelmoinnissa.

4.3.2. Ilmaisuvoima

Termiä *ilmaisuvoima* käytetään monissa eri merkityksissä. Matthias Felleisenin formaalin määritelmän mukaan ohjelmointikieli A on ilmaisuvoimaisempi kuin kieli B, jos A:lla voi kirjoittaa ohjelman, jota ei voi kirjoittaa B:llä muuttamatta ohjelman globaalia rakennetta. [33]

Usein ilmaisuvoimalla kuitenkin tarkoitetaan myös kirjoitetun ohjelman lukemisen helppoutta; siis sitä, kuinka hyvin ohjelmakoodi kuvaa sen päämäärää eikä pelkästään suoritusohjeita tietokoneelle. Tämä liittyy oleellisesti deklarativisuuteen, mikä on tärkeä osa useimpia funktionaalisia ohjelmointikieliä.

Clojuressa hyvä esimerkki deklarativisuudesta ja ilmaisuvoimasta on listojen funktionaalinen käsittely. Ohjelmissa 4.4, 4.5, 4.6 ja 4.7 on esitetty sekä Javan versioilla 7 ja 8, Clojurella että Scalalla ohjelmakoodi, joka käy läpi listan henkilöitä ja palauttaa listan vanhoista henkilöistä järjestettynä iän mukaan. Henkilöt on kuvattu kaikissa ohjelmissa siten kuin edellisen luvun ohjelmissa 4.1, 4.2 sekä 4.3 kullekin ohjelmointikielelle esitettiin.

Ohjelmista näkyy selvästi, miten Java 7:lla kirjoitettu koodi on ennemminkin tietokoneelle kirjoitettu ohje siitä, miten lista käydään läpi ja uusi lista muodostetaan, kun taas Clojurella tai Scalalla kirjoitetussa koodissa ilmaistaan vain että ohjelmoija haluaa suodattaa listasta tietyt alkiot, ottamatta kantaa siihen miten varsinainen to-

teutus toimii. Java 8:lla kirjoitettu koodi on jo lähempänä Clojure- tai Scala-koodia, mutta sekin sisältää enemmän tietokoneelle suunnattuja ohjeita.

Ohjelma 4.4: Esimerkkiohjelma Java 7:lla

```
1 public bool isOld(Person p) {
2     return p.getAge() > 60;
3 }
4
5 public List<Person> filterPeople(List<Person> people) {
6     List<Person> oldPeople = new ArrayList<>();
7     for(Person p : people) {
8         if(isOld(p)) {
9             oldPeople.add(p);
10        }
11    }
12    return Collections.sort(oldPeople, new Comparator<Person>() {
13        public int compare(Person a, Person b) {
14            return a.getAge().compare(b.getAge());
15        }
16    });
17 }
```

Ohjelma 4.5: Esimerkkiohjelma Java 8:lla

```
1 public bool isOld(Person p) {
2     return p.getAge() > 60;
3 }
4
5 public List<Person> filterPeople(List<Person> people) {
6     return people.stream()
7         .filter(p -> isOld(p))
8         .sorted(Comparator.comparing(Person::getAge))
9         .collect(toList());
10 }
```

Ohjelma 4.6: Esimerkkiohjelma Clojurella

```
1 (defn old? [p]
2   (> (:age p) 60))
3
4 (defn filter-people [people]
5   (sort-by :age (filter old? people)))
```

Ohjelma 4.7: Esimerkkiohjelma Scalalla

```
1 def isOld(p: Person) = p.age > 60
2
3 def filterPeople(people: List[Person]) = people.filter(isOld).sortBy
   (p => p.age)
```

Scala- ja Java-ohjelmia vertailemalla käy myös ilmi, miten eri tavoilla staattisen tyyppityksen voi toteuttaa. Scalassa on voimakas tyyppipäätelyjärjestelmä, joka vähentää tarvetta merkitä muuttujien ja funktioiden tyyppisiä näkyviin, jos kääntäjä pystyy päättämään ne. Esimerkiksi ohjelmassa 4.7 funktioiden paluuarvoja (`Boolean` ja `List[Person]`) ei ole kirjoitettu näkyviin, koska kääntäjä tietää mitä tyyppiä `>`-, `filter`- ja `sortBy`-funktioiden paluuarvot ja siten `isOld`- ja `filterPeople`-funktioiden paluuarvot ovat. Kehitysympäristöt osaavat myös tarvittaessa näyttää merkitsemättä jätetyt tyypit. Scalassa yleensä riittää, että funktion parametrien tyyppi on merkitty. Vain harvoissa tapauksissa kääntäjä ei onnistu päättämään paluutyyppejä, jolloin ohjelmoijan on merkittävä se näkyviin. Yleisin tällainen tapaus Scalassa on rekursiivinen funktio. Myös Haskellista löytyy vastaava, voimakas tyyppipäätelyjärjestelmä. Haskell vaatii yleensä vielä vähemmän tyyppien merkitsemistä näkyviin kuin Scala. Lähes aina myös parametrien tyypit ja rekursiivisten funktioiden palauttamien tyypit voi jättää merkitsemättä näkyviin. Selkeyden vuoksi on kuitenkin usein tapana merkitä ainakin osa tyypeistä.

Ohjelmointikielen ilmaisuvoimasta haastatteluissa eniten esiinnoussut asia oli se, millaisia kokonaisuuksia ohjelmakoodi muodostaa. Ilmaisuvoimaisemmalla kielellä syntyy luonnostaan pienempiä kokonaisuuksia. Tällöin ohjelmoijan on helpompi hallita ohjelman rakennetta. Pienemmät kokonaisuudet on myös helpompi hahmottaa

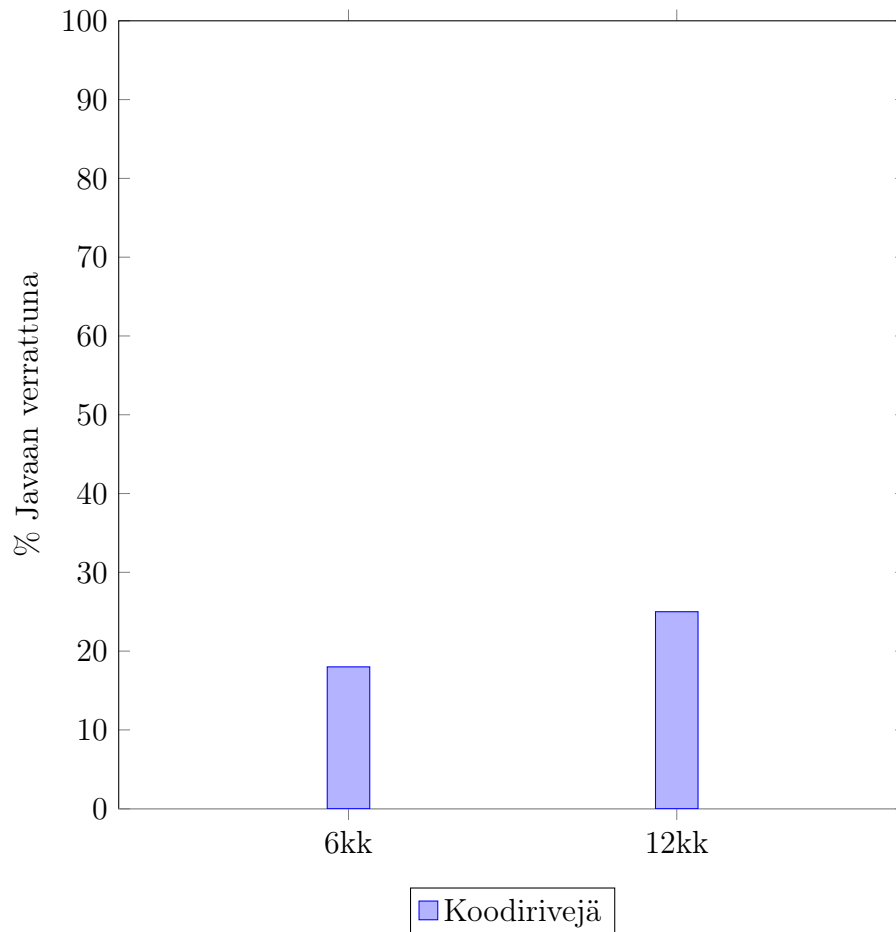
ja pitää mielessä. Tekemällä funktioista tiiviitä ja nimeämällä ne hyvin ohjelmoijan ei tarvitse muistaa ulkoa laajoja kokonaisuuksia ja niiden toimintaa. Tällä saavutetaan korkeampi tuottavuus erityisesti kun useampi ohjelmoija työskentelee saman koodin parissa. Tietyn kokonaisuuden ottaminen työn alle vaatii aina sen sisäistämisen ennen kuin muutosten tekeminen on mahdollista, ja deklaratiiivisen koodin hahmottaminen on nopeampaa ja helpompaa. [34]

Käyttöliittymäpään ohjelmoinnissa ilmaisuvoima nousi esiin funktionaalisen reaktiivisen ohjelmoinnin yhteydessä (*Functional Reactive Programming, FRP*). Funktionaalisen reaktiivisen ohjelmoinnin nähtiin helpottavan tapahtumien käsittelyä, kun se voitiin tehdä funktionaalisella puhtaalla tavalla. Ohjelmoijan ei itse tarvinnut aiheuttaa sivuvaikutuksia, vaan kaikki itse kirjoitettu ohjelmakoodi oli edelleen puhtaan funktionaalista, ja sivuvaikutusten käsittely jäi käyttöliittymäkirjaston huo- leksi.

Funktionaalisen reaktiivisen ohjelmoinnin sanottiin myös helpottavan tilan välittämistä käyttöliittymäkomponenteille. Samoin kuin tapahtumien käsittely, myös datan käsittely voitiin FRP:n avulla tehdä puhtaasti, ja tilan hallinta jäi käyttöliittymäkirjaston vastuulle. Sekä tilan tallentaminen että tapahtumien käsittely nähtiin ongelmana perinteisessä funktionaalisessa ohjelmoinnissa. Erityisesti haastateltavat, joille reaktiivinen funktionaalinen ohjelmointi ei ollut tuttu, esittivät ne huolenaiheena. Suurin osa haastateltavista ei ollut tutustunut kyseiseen paradigmaan. Ne, jotka olivat, pitivät sitä kuitenkin erinomaisena ratkaisuna epäpuhtaan reaktiivisen käyttöliittymän toteuttamiseen.

Ilmaisuvoimaa ja selkeää rakennetta kommentoitiin myös kyselytutkimuksen avoimissa vastauksissa. Äärimmäisenä esimerkkinä niiden tärkeydestä toimii tämä lainaus: *"Ilman Clojurea tämä laiva olisi uponnut matkalla moneen kertaan. Javalla koodimäärä olisi ollut niin paljon suurempi ja muutoksien tekeminen olisi käynyt niin tuskalliseksi, että emme olisi ikinä onnistuneet niinkään hyvin kun onnistuimme."*

Koodirivien määrässä mitattunakin Clojure-koodi on huomattavasti Java-koodia tiiviimpää. Kyselytutkimukseen annetut vastaukset koodin ilmaisuvoiman osalta on



Kuva 4.4: Kyselytutkimukseen osallistuneiden arvio Clojuren ilmaisuvoimasta koodiriveissä mitattuna Javaan verrattuna

esitetty kuvassa 4.4. Jälkimmäisessä kyselyssä vastaajat arvioivat Clojure-koodin pituuden olevan vain neljäsosan vastaavan Java-koodin pituudesta.

4.3.3. Aiemmat tutkimukset

Myös aikaisemmin tehdyt tutkimukset tukevat tuloksia funktionaalisen ohjelmoinnin tehokkuuden osalta. Erann Gat[35] ja Lutz Prechelt[36] vertailivat ohjelmoinnin tuottavuutta eri kielillä ja havaitsivat, että Lisp-ohjelmoija käytti keskimäärin 50 % vähemmän aikaa saman ohjelman tekemiseen kuin Java-ohjelmoija. Lisp-ohjelman rivimäärä oli keskimäärin 43 % vastaavan Java-ohjelman rivimäärästä. Tutkimukseen osallistuneiden Lisp-ohjelmoijien kokemus oli hieman Java-ohjelmoijien koke-
musta matalampi.

Ericsson-yhtiössä havaittiin jopa nelinkertainen muutos tehokkuuden kasvussa ja virheiden vähenemisessä, kun siirryttiin imperatiivisesta ohjelmoinnista funktionaa-

liseen. [37] Tämän tutkimuksen osalta on kuitenkin huomioitava, että Erlang kehitettiin nimenomaan kyseiseen käyttötarkoitukseen, joten osa muutoksesta selittyy kielen ominaisuuksilla eikä ohjelmointiparadigmalla.

4.4. Funktionaalisen ohjelmointikielen vaikutukset ohjelmistoprojektiin

Itse ohjelmistoprojektiin kielivalinnalla ei nähty olevan suurta vaikutusta. Haastatteluissa osallistujilta kysyttiin näkemyksiä funktionaalisten kielten ekosysteemistä, työkaluista sekä vaikutuksista projektinhallintaan.

4.4.1. Ekosysteemi

Yksi haastattelujen teemoista oli frameworkien ja kirjastojen saatavuus. Koska Clojure-koodista on mahdollista kutsua Java-koodia, on mikä tahansa Java-kirjasto käytettävissä myös Clojure-ohjelmassa. Ainoana ongelmana haastatteluissa mainittiin se, että Java-kirjaston käyttäminen Clojuressa on jossain määrin kankeampaa kuin Clojure-kirjastojen. Asian katsottiin johtuvan siitä, että Java-kirjastot toimivat usein imperatiivisesti, jolloin niiden käyttö vaatii funktionaalista periaatteista poikkeamisen. Clojure kuitenkin mahdollistaa myös imperatiivisten kirjastojen käytön funktionaalisisessa kontekstissa. Haastateltavista kaksi kertoi myös tehneensä Java-kirjaston ympärille oman Clojure-kirjaston, joka kätkee sisälleen imperatiivisen ohjelman ja tarjoaa Clojuressa helpommin käytettävän funktionaalisen rajapinnan Java-kirjaston toiminnallisuudelle. Samat huomiot pätevät myös Scalaan Java-kirjastojen osalta ja F#:iin C#-kirjastojen osalta.

Haskelliin tutustunut haastateltava piti sitä huonona vaihtoehtona yrityskäyttöön kunnollisen ekosysteemin puuttumisen vuoksi. Haastateltava oli itse opiskellut Haskellia oman osaamisensa ja ajatusmaailmansa kehittämisen vuoksi, mihin hän piti sitä erittäin hyvänä työkaluna.

Ongelmatilanteissa imperatiivinen ohjelmointi kuitenkin loistaa funktionaaliseen verrattuna. Yleisyyden vuoksi apua on saatavilla paljon laajemmin imperatiivisiin kuin funktionaalisiin kieliin. Jos funktionaalisella kielellä törmää johonkin harvinaiseen ongelmaan, vastauksen tai ohjeiden löytäminen Googella voi olla todella han-

kalaa. Myös kielen antamat virheilmoitukset ovat monissa tapauksissa ongelmallisia. Varsinkin laiska laskenta aiheuttaa virhetilanteita, joissa virhe ilmenee aivan eri paikassa kuin missä se todellisuudessa ohjelmakoodissa on. Tämä hankaloittaa huomattavasti virheen paikallistamista ja korjaamista.

4.4.2. Työkalut

Kaikki haastateltavat kertoivat käyttävänsä Clojure-ohjelmointiin samaa ohjelmointiympäristöä kuin Java-ohjelmointiinkin. Kaikilla haastateltavilla oli käytössä joko *Eclipse*- tai *IntelliJ IDEA* -ohjelmointiympäristö. Kumpikin on laajassa käytössä, ja molempiin on saatavilla Clojure-ohjelmointia helpottava lisäosa. Varsinaisen ohjelmiston osalta haastateltavat eivät nähneet eroa Java- ja Clojure-ohjelmoinnin välillä.

Ohjelmointityökaluihin liittyy kuitenkin yksi asia, jonka haastateltavat kokivat huomattavasti paremmaksi Clojure-ohjelmoinnissa. Clojure-ohjelmointiin kuuluu hyvin oleellisena osana *REPL* eli vuorovaikutteinen tulkki. Se on ohjelmointiympäristön osa, jossa voi suorittaa kirjoittamaansa koodia ja nähdä suorituksen tuloksen välittömästi. *REPL*in avulla koodin kirjoittaminen nopeutuu huomattavasti, kun tekemänsä koodin toimintaa pystyy koko ajan testaamaan erilaisilla syötteillä. Ohjelmoijan on myös virhetilanteissa helppo selvittää, missä kohtaa virhe tapahtuu. Useilla haastateltavilla *REPL*in käyttö onkin korvannut Java-ohjelmoinnissa yleisen debuggerin käytön lähes kokonaan.

REPL on yleensä integroitu kiinteästi ohjelmointiympäristöön, ja kirjoitetun koodin lataaminen siihen on helppoa. Kun koko ohjelman suoritus tapahtuu *REPL*issä, on ohjelmakoodin muokkaaminen mahdollista käynnistämättä koko ohjelmaa uudelleen, mikä myös nopeuttaa kehityssykliä. Java-kehityksessä se ei yleensä ole mahdollista tai vaatii erityisesti sitä varten kehitetyn työkalun. *REPL* myös mahdollistaa esimerkiksi yksittäistenkin testien ajamisen helposti ja nopeasti suoraan kehitysympäristössä. Haastatteluissa kerrottiin tämän auttavan erityisesti testivetoisessa kehityksessä (*Test-driven design*, *TDD*).

4.4.3. Projektinhallinta

Haastatteluun pyydettiin myös yksi Clojure-projektia johtanut projektipäällikkö. Osana tutkimusta haluttiin selvittää, millaisia vaikutuksia teknologiavalinnoilla on yleisesti projektityöskentelyyn ja projektinhallintaan. Samaa asiaa kysyttiin kaikilta haastateltavilta, mutta se oli suunnattu erityisesti projektipäällikön roolissa toimineelle.

Projektipäällikkö ei kokenut, että teknologiavalinta olisi erityisesti vaikuttanut projektinhallintaan. Suurin vaikutus sillä tuntui olevan projektitiimin yhteishenkeen. Projektin alkuvaiheissa monille uusi teknologia aiheutti tavallista isompia murheita tekijöille, mikä mahdollisesti laski tiimin moraalialue. Toisaalta osa haastateltavista koki, että yhdessä uuden oppiminen ja kehittyminen toi tiimiä tavallista paremmin yhteen. Clojure-mentorien esimerkki koettiin hyvänä eteenpäin vievänä asiana.

Projektin ylläpitovaiheeseen siirtyminen aiheutti myös huolta. Projektipäällikkö ja osa muista haastateltavista esitti huolen, että ylläpitoon siirtyneelle projektille ei välttämättä löydy osaavia tekijöitä, jos Clojure-osaajat ovat kiinni uudessa aktiivisessa projektissa, ja Clojuria osaavat tekijät ovat harvassa. Tällöin tekijöiltä vaadittaisiin useaan projektiin osallistumista yhtä aikaa. Mahdottomuutena tilannetta ei nähty. Kukaan haastateltavista ei uskonut, että Clojure-tekeminen loppuisi täysin, vaan että osaavia Clojure-ohjelmoijia kyllä tulee jatkossakin löytymään eivätkä ylläpitovaiheeseen siirtyneet projektit jää hakoteille.

5. YHTEENVETO

Funktionaalinen ohjelmointi ratkaisee monia perinteisiä ohjelmoinnin ongelmia. Funktionaalinen ohjelmointi helpottaa ohjelman luettavuutta, vähentää kompleksisuutta ja lisää ilmaisuvoimaa. Funktionaalisen ohjelman automaattinen testaaminen on helpompaa, ja funktionaalisen ohjelmoinnin periaatteet soveltuvat hyvin web-ohjelmointiin.

Web-palvelinohjelmassa suuri osa työtä on datan käsittelyä. Funktionaalisen ohjelmoinnin datankäsittelyominaisuuksilla ja funktioiden koostamisella saadaan aikaan tehokas, muokattava ja helposti ymmärrettävä liukuhihna, jolla dataa saadaan siirrettyä tietokannasta palvelimen rajapintaan ja toiseen suuntaan.

Web-käyttöliittymä puolestaan on luonnostaan reaktiivinen. Sen toiminta perustuu käyttäjän syötteisiin sekä palvelimelta tulevaan dataan reagointiin. Funktionaalinen reaktiivinen ohjelmointi soveltuu luontaisesti hyvin kyseiseen toimintaperiaatteeseen. Myös käyttöliittymän tapahtumista sekä arvoista saadaan aikaiseksi liukuhihnoja, joilla kulkevia tapahtumia ja arvoja käsitellään funktionaalisilla työkaluilla.

Tutkimuksesta käy selvästi ilmi, että funktionaalisen ohjelmoinnin edut web-ohjelmistokehityksessä ovat huomattavat. Ohjelmoijien tuottavuus on parempi, ja syntyvä ohjelmakoodi on laadukkaampaa kuin imperatiivisilla ohjelmointikielillä. Tutkimukseen osallistuneet ohjelmoijat myös tekevät mieluummin töitä funktionaalisella kuin imperatiivisella ohjelmointikielellä.

Jos funktionaalinen ohjelmointi on selkeästi imperatiivista ohjelmointia parempi vaihtoehto, miksei se ole nykyistä laajemmassa käytössä? Hyvässäkin mallissa on toki riskinsä. Imperatiivinen paradigma on paljon käytetympi kuin funktionaalinen. Valmiiksi paradigman osaavia tekijöitä on tarjolla enemmän ja apua ongelmatilanteisiin on tarjolla enemmän. Työntekijöiden kouluttaminen uuteen paradigmaan ei ole ilmaista, ja toiset oppivat funktionaalisen ohjelmoinnin helpommin kuin toiset. Jotta uusia tekijöitä saadaan koulutettua, täytyy projekteissa olla mukana osaavia

mentoreita, joita on hyvin rajallinen määrä. Myös osaamisen jatkuvuus on riskitekijä. Jos teknologia hylätään muutaman projektin jälkeen, on odotettavissa ongelmia kun projektit siirtyvät ylläpitoon mutta osaavia tekijöitä ei enää löydykään.

Kaikenkaikkiaan funktionaaliseen ohjelmointiin panostaminen pienentää monia riskejä. Mitä useampia projekteja funktionaalisella kielellä toteutettuja projekteja on, sitä useampia mentoriksi kykeneviä henkilöitä yritykseen saadaan. Samoin riski ylläpidon epäonnistumisesta pienenee. Kun projekteja on useita, myös projektin perustamisen kustannukset pienenevät kun yhteiskäyttöisiä menetelmiä ja ohjelmakoodia voidaan hyödyntää.

Tämän tutkimuksen pohjalta suosittelen ehdottomasti funktionaalisen ohjelmoinnin hyödyntämistä web-ohjelmistokehityksessä, ja kannustan Solita Oy:tä käyttämään Clojurea ja Clojurescriptiä myös tulevaisuuden ohjelmistoprojekteissa.

LÄHTEET

- [1] Antero Taivalsaari, Tommi Mikkonen, Matti Anttonen, and Arto Salminen. The death of binary software: End user software moves to the web. In *Proceedings of the 2011 Ninth International Conference on Creating, Connecting and Collaborating Through Computing, C5 '11*, pages 17–23, Washington, DC, USA, 2011. IEEE Computer Society.
- [2] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989.
- [3] Anttijuhan Lantto. Funktionaalinen reaktiivinen ohjelmointi web-sovelluksissa. Master's thesis, Helsingin yliopisto, Helsinki, 2015.
- [4] Michel Schinz and Martin Odersky. Tail call elimination on the java virtual machine. *Electronic Notes in Theoretical Computer Science*, 59(1):158 – 171, 2001.
- [5] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.
- [6] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ICFP '98*, pages 129–140, New York, NY, USA, 1998. ACM.
- [7] Martin Odersky. *Scala By Example*. Programming Methods Laboratory, Lausanne, Switzerland, 1st edition, 2014.
- [8] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.
- [9] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932.

- [10] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.
- [11] Raúl Rojas. A tutorial introduction to the lambda calculus. *CoRR*, abs/1503.09060, 2015.
- [12] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, August 1978.
- [13] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, September 1993.
- [14] Paul Graham. *Hackers and Painters: Essays on the Art of Programming*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.
- [15] Mike Gordon. From LCF to HOL: A short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.
- [16] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
- [17] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [18] Martin Odersky. The Scala experiment – can we provide better language support for component systems? In *Proc. ACM Symposium on Principles of Programming Languages*, pages 166–167, 2006.
- [19] F# historical acknowledgements, URL: <http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ack.aspx>, 2012.
- [20] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. <http://haskell.org/>, September 2002.
- [21] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

- [22] Charles Severance. JavaScript: Designing a language in 10 days. *Computer*, 45(2):7–8, February 2012.
- [23] C. Saternos. *Client-Server Web Apps with JavaScript and Java*. O'Reilly, 2014.
- [24] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages (PADL'02)*, January 2002.
- [25] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.
- [26] S. Blackheath and A. Jones. *Functional Reactive Programming*. Manning, 2016.
- [27] Bacon.js, URL: <https://baconjs.github.io/>, 2013.
- [28] re-frame, URL: <https://github.com/Day8/re-frame>, 2015.
- [29] Anita Saaranen-Kauppinen and Anna Puusniekka. *KvaliMOTV - Menetelmäopetuksen tietovaranto*, URL: <http://www.fsd.uta.fi/menetelmaopetus/>. Yhteiskuntatieteellinen tietoarkisto, Tampere, 2006.
- [30] P. Hannila and P. Kyngäs. Teemahaastattelu laadullisessa tutkimuksessa. Bachelor's thesis, Helsingin ammattikorkeakoulu Stadia, Helsinki, 2008.
- [31] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [32] Metosin Oy. Compojure-API, URL: <https://github.com/metosin/compojure-api>, 2014.
- [33] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.
- [34] Steve Vinoski. Welcome to "the functional web". *IEEE Internet Computing*, 13(2):102–104, 2009.

- [35] Erann Gat. Point of view: Lisp as an alternative to Java. *Intelligence*, 11(4):21–24, December 2000.
- [36] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000.
- [37] Ulf Wiger and Ericsson Telecom Ab. Four-fold increase in productivity and quality - industrial-strength functional programming in telecom-class products, 2001.