TURUN YLIOPISTO

Funktionaalinen ohjelmointi: matemaattinen tausta, erikoisominaisuudet ja niiden käyttö yleisemmissä kielissä

Tietotekniikan laitos, Teknillinen tiedekunta

Laatija:
Niko Jokela

Ohjaajat:
Jari-Matti Mäkelä
Anne-Maarit Majanoja

Toukokuu 2024

Kandidaatintyö
Tietotekniikan laitos, Teknillinen tiedekunta
Turun yliopisto

Oppiaine: Tietoliikenne- ja kyberturvallisuusteknologia
Tutkinto-ohjelma: Tietotekniikka
Tekijä: Niko Jokela
Otsikko: Funktionaalinen ohjelmointi: matemaattinen tausta, erikoisominaisuudet ja niiden käyttö yleisemmissä kielissä
Sivumäärä: 25 sivua, 0 liitesivua
Päivämäärä: Toukokuu 2024

Funktionaalinen ohjelmointi on erilainen, ja vähemmän suosittu ohjelmoinnin ajattelutapa vallitsevaan imperatiiviseen olio-ohjelmointiin verrattuna. Funktionaalisia ohjelmointitapoja on viime vuosina kuitenkin alettu käyttämään yleisimmissä ohjelmointikielissä. Tässä työssä tarkastellaan ristiriitaa sen suhteen, miksi puhtaat funktionaaliset kielet eivät ole yleisessä suosiossa, mutta joitain niiden ominaisuuksia käytetään muissa kielissä laajalti. Asiaa lähestytään tutustumalla funktionaaliseen ohjelmointiin, ja vastataan kolmeen tutkimuskysymykseen: mikä on funktionaalisen ohjelmoinnin matemaattinen tausta? Mitä erikoisominaisuuksia sillä, ja erityisesti käytetyimmällä kielellä Haskellilla, on? Miten kyseisiä ominaisuuksia käytetään yleisimmissä ohjelmointikielissä Pythonissa ja JavaScriptissä?

Tutkimus on tehty kirjallisuuskatsauksen tavoin. Tärkeimmät lähteet ovat Bartosz Milewskin ”Category theory for programmers”-kirja, ja Haskellia käsittelevät kirjat “Haskell Programming from first principles” ja “Practical Haskell, A Real World Guide to Programming”. Yleisempiä ohjelmointikieliä Pythonia ja JavaScriptiä käsittelevässä luvussa ja aiheessa on käytetty apuna niiden (puoli)virallista dokumentaatiota.

Funktionaalisen ohjelmoinnin matemaattinen tausta perustuu lambdakalkyyliin ja kategoriateoriaan. Sen erikoisominaisuuksia ovat mm. korkea abstraktiotaso, puhtaat funktiot, laiska suoritustapa, lambdakalkyylista kumpuavat ohjelmointitavat, sekä sivuvaikutusten hallinta kategoriateoriasta tutuilla monadeilla. Yleisemmissä kielissä käytetään erityisesti funktionaalisesta ohjelmoinnista tuttuja, deklaratiivisia iterointimenetelmiä, sekä anonyymejä funktioita. Ideaa monadeista on sovellettu asynkronisten funktiokutsujen käsittelyyn. Tutkielman lopussa mietitään, voisiko funktionaalinen ohjelmointitapa tulla tulevaisuudessa yleisemmäksi.

Asiasanat: funktionaalinen ohjelmointi, Haskell, lambdakalkyyli, kategoriateoria

Sisällys

1 Johdanto

Funktionaalinen ohjelmointi on erilainen, ja vähemmän suosittu, ohjelmoinnin ajattelutapa vallitsevaan olio-ohjelmointiin verrattuna. Kyselytutkimuksen [1] mukaan käytetyimmät ohjelmointikielet ohjelmistokehittäjien keskuudessa ovat oliokieliä, jotka tukevat myös muita ajattelumalleja. Näihin kuuluvat mm. JavaScript/TypeScript, Python, Java, C# ja C++. Suosituinta puhtaasti funktionaalista kieltä Haskellia käyttää samaisen tilaston mukaan vain 2,09 % kehittäjistä. Funktionaalisen ohjelmoinnin ominaisuuksia käytetään nykyään kuitenkin laajalti yleisemmissä kielissä.

Työn aiheen valintaa on inspiroinut käymäni web-ohjelmoinnin kurssi. Siinä käytettiin paljon funktionaalisesta ohjelmoinnista vaikutteita ottavaa JavaScriptille tehtyä React-kirjastoa web-sivujen tekemiseen. Itse funktionaalinen ohjelmointi jäi kyseisellä kurssilla kuitenkin hieman salaperäiseksi. Tarkoituksenani tässä työssä on tuoda funktionaalinen ohjelmointi tutummaksi vastaamalla kolmeen tutkimuskysymykseen:

TK 1: Mikä on funktionaalisen ohjelmoinnin matemaattinen tausta (luku 2).
TK 2: Mitä erikoisominaisuuksia funktionaalisella ohjelmoinnilla, erityisesti suosituimmalla puhtaalla funktionaalisella kielellä Haskellilla, on (luku 3).
TK 3: Mitä ja miten funktionaalisia ominaisuuksia käytetään yleisemmissä kielissä (luku 4).

Tutkimus on tehty kirjallisuuskatsauksen tavoin. Tutkimuskysymys 3 perustuu pitkälti Reactissa vastaan tulleisiin ja tunnistettaviin funktionaalisiin ohjelmointitapoihin. Eniten käytettyjä lähteitä olivat Bartosz Milewskin ”Category theory for programmers”-kirja ja kirjaan perustuva web-sivu, johon lähteet viittaavat. Kirja on hyvä ja kiitetty välimalli, jossa kategoriateoria esitetään helpommin lähestyttävällä tavalla ohjelmoijille. Matematiikan opukset kategoriateoriasta osoittautuivat käytännössä lukukelvottomaksi aiheesta alun perin mitään ymmärtämättömälle. Haskellia käsittelevät kirjat “Haskell Programming from first principles” ja “Practical Haskell, A Real World Guide to Programming” olivat yhdessä varmasti eniten käytettyjä lähteitä tässä työssä. Luvussa 4 päälähteenä olivat Mozillan (puoli)virallinen JavaScript-opas ja Pythonin dokumentaatio. Mainittakoon, että moderneista, nopeasti kehittyvistä ohjelmointikielistä, on paikoin vaikea löytää virallista dokumentaatiota.

Tutkimuskysymystä 1 käsitellään luvussa 2 ”Funktionaalisen ohjelmoinnin matemaattinen tausta”. Sitä tarkastellaan lähtien liikkeelle tietojenkäsittelytieteen historiasta. Ensimmäiset ideat tietokoneen käsitteestä ovat 1600-luvun lopulta. Ideat alkoivat konkretisoitumaan 1900-luvun alussa Alan Turingin ja Alonzo Churchin läpimurtojen takia. Luvussa kerrotaan, miten matemaattinen tausta perustuu Alonzo Churchin keksimään lambdakalkyyliin ja abstraktin algebran osa-alueeseen kategoriateoriaan. Tutkimuskysymystä 2 käsitellään luvussa 3 ”Funktionaalinen ohjelmointi ja Haskell”. Siinä esitellään ensin yleisin puhtaasti

funktionaalinen ohjelmointikieli Haskell, ja perehdytään sen hieman epätavalliseen laiskaan suoritustapaan. Sen jälkeen esitellään viime kädessä lambdakalkyylista ja kategoriateoriasta johtuvia ohjelmoinnin erikoisominaisuuksia ja tapoja. Tutkimuskysymystä 3 käsitellään luvussa 4 ”Funktionaalisuus oliokielissä”. Siinä esitellään, mitä ja miten luvussa 3 esitettyjä ominaisuuksia käytetään yleisemmissä kielissä. Vertailukielinä käytetään kahta yleisintä ohjelmointikieltä JavaScriptia ja Pythonia. Luvussa 5 tehdään loppupäätelmät työn aiheista, ja keskustellaan ohjelmoinnin tulevaisuudesta.

2 Funktionaalisen ohjelmoinnin matemaattinen tausta

Funktionaalisen ohjelmoinnin käsitteen ymmärtäminen voi olla vaikeaa ilman käsitystä tietokoneiden ja tietojenkäsittelytieteen historiasta. Tässä luvussa tutustutaan funktionaalisen ohjelmoinnin matemaattiseen taustaan lähestymällä sitä tästä näkökulmasta. Luvussa esitellään muodollisia matemaattisen laskennan malleja Turingin konetta, ja lambdakalkyylia. Lambdakalkyyliin paneudutaan tarkemmin, käsitellään sen jatkokehitystä ja sen ideoiden linkittymistä abstraktin algebran osa-alueeseen kategoriateoriaan.

2.1 Tietojenkäsittelytieteen historia

Kuuluisaa matemaatikkoa ja tiedemiestä Gottfried Leibnizia (1646–1716) voidaan pitää ensimmäisenä tietojenkäsittelytieteilijänä. Hänellä oli kaksi aiheeseen liittyvää ajatusta. Voidaanko luoda universaali kieli, jolla kaikki mahdolliset matemaattiset ongelmat voidaan esittää? Voidaanko löytää keino ratkaista kaikki ongelmat, joita kyseisellä kielellä voi esittää? Jälkimmäistä kysymystä kutsutaan myös nimellä Entscheidungsproblem – päätäntäongelma. Voiko tietokone etukäteen kertoa, ratkeaako laskettava ongelma äärellisessä ajassa? [2, luku 1]

Universaalin kielen voi rakentaa (ensimmäisen kertaluvun) predikaattilogiikalla, jossa käytetään mm. kvanttoreita, konnektiiveja ja muuttujia mallintamaan luonnollisen kielen lauseita. Päätäntäongelman tutkiminen osoittautui kullanarvoiseksi tietojenkäsittelytieteen kannalta: vuonna 1936 Alonzo Church ja Alan Turing erikseen osoittivat päätäntäongelman mahdottomaksi. Todistusta varten heidän piti luoda muodollinen malli matemaattiselle laskennalle. Turing keksi nimeään kantavan koneen, joka suorittaa ohjelmaa muuttamalla koneen tilaa käskyjen mukaan. Turingin konetta voidaan pitää tietokoneen abstraktina mallina. [2, luku 1]

Alonzo Church keksi lambdakalkyylin (lambda calculus, myös nimellä lambdalaskenta), jolla on mahdollista muodollistaa matemaattista laskentaa funktioiden avulla. Funktiolla tarkoitetaan suhdetta sille annettavien alkuarvojen ja ulostulon välillä [3, s. 30–32]. Funktion arvon laskenta etenee sieventämällä sitä annetuilla säännöillä yksinkertaisemmaksi. Lambdakalkyylia voidaan pitää ohjelmointikielen abstraktina mallina ja sitä käsitellään luvussa 2.3. Turing osoitti kummankin em. laskentamallin olevan teoreettisella tasolla yhtä kykeneviä mallintamaan matemaattisen lausekkeen ratkaisua. Niitä voidaan pitää ekvivalentteina, ja Turingin koneet ja lambdakalkyylilla tehdyt ohjelmat voidaan kääntää toisikseen. [2, luku 1]

2.2 Tietokonearkkitehtuuri ja ohjelmointikielet

Nykyään ylivoimaisesti yleisin [4] Von Neumannin tietokonearkkitehtuuri perustuu Turingin koneen konseptiin, jossa ongelma ratkaistaan käskyttämällä tietokonetta yksiselitteisesti vaihe vaiheelta käyttämällä ohjelmointikielen komentoja. Ohjelmointikieliä, jotka noudattavat tätä Turingin koneen kaavaa, kutsutaan

imperatiivisiksi. Näitä kieliä ovat mm. kaikki yleisimmät korkean tason kielet, kuten Python ja JavaScript. Korkean tason ohjelmointikielillä tarkoitetaan kieliä, jotka merkittävästi abstrahoivat tietokoneen yksityiskohtaista toimintaa. Niitä on suunniteltu ihmisille helpompikäyttöiseksi ja -lukuiseksi. [5] Korkean tason kielet täytyy kääntää imperatiivisiin matalan tason Assembly-kieliin. Niistä ne käännetään edelleen tietokoneen ymmärtämään nollista ja ykkösistä koostuvaan konekieleen. [2, luku 1]

Funktionaaliset ohjelmointikielet sen sijaan perustuvat lambdakalkyyliin. Funktionaalisen kielen kenties päälle näkyvin ero muihin verrattuna on idea deklaratiivisesta eli kuvailevasta ohjelmoinnista. Tietokoneen käskyttämisen sijaan sille kuvaillaan, mitä halutaan laskea, eikä yksityiskohtia siitä miten, sen pitäisi tapahtua. [6, luku 11] Korkean tason funktionaaliset kielet käännetään jollain sievennyskoneella (reduction machine) yksinkertaisimpaan muotoonsa. Tietokone, joka pystyy käyttämään tätä muotoa toimintaansa varten, vaatii omanlaisensa, Von Neumannista poikkeavan arkkitehtuurin [7]. Kyseiset tietokonearkkitehtuurit ovat tällä hetkellä lähinnä tutkimuskäyttöä varten ja funktionaaliset kielet pitää lopulta erikseen kääntää Assembly-kielelle, jotta ne toimisivat ”tavallisissa” tietokoneissa. [8, s. 288–290]

Imperatiivisen kielen ohjelmointi perustuu Turingin koneen tavoin jonkin ohjelmalle yhteisen, globaalin tilan muuttamiseen. Lambdakalkyyliin perustuva funktionaalinen ohjelmointi perustuu matemaattisiin ilmaisuihin. Kenties perustavanlaatuisin eroavaisuus edellä mainittujen ohjelmointitapojen välillä on käsitys ohjelmakoodin sivuvaikutuksista. Matemaattisesti puhdas funktio palauttaa aina saman arvon samoilla alkuarvoilla. Funktio, joka muuttaa jotain yhteistä tilaa, ei aina palauta samaa arvoa samoilla alkuarvoilla. Palautusarvo riippuu siitä, mikä yhteisen tilan alkuarvo on ollut. Jos funktio muuttaa jotain ohjelmalle yhteistä asiaa, sitä kutsutaan sen sivuvaikutukseksi. Sivuvaikutukselliselle funktiolle on oma terminsä ”rutiini” (routine). Funktionaalisen kielen funktiot eivät oletusarvoisesti aiheuta sivuvaikutuksia. Sivuvaikutuksiksi lasketaan kuitenkin myös tietokoneen käytön kannalta hyvin olennaisia asioita, kuten vuorovaikutus sen käyttäjän tai esimerkiksi verkkoliikenteen kanssa. [9, s. 5] Funktionaalisissa kielissä on mahdollista käsitellä sivuvaikutuksia, mutta se täytyy tehdä tietyllä tavalla. Asiaa käsitellään tässä työssä luvussa 3.6.

2.3 Lambdakalkyyli

Lambdakalkyyli on muodollinen laskennan malli, jolla voi ilmaista mitä tahansa matemaattisia laskuongelmia Turingin koneen tavoin. Sen nimi tulee hieman sattumanvaraisesti kreikan kielen kirjaimesta λ. Se on perusperiaatteeltaan hyvin yksinkertainen kieli, jonka ilmaisu koostuu vain kolmesta termistä:

- muuttujat (esim. x, y, z)
- funktion määritys käyttäen symboleita lambda (λ) ja ”.” (esim. `λx.x+1` vastaa matemaattista funktiota `f(x) = x + 1`)
- funktion sovellus (esim. `(λx.x+1)1` vastaa matemaattista funktiota `f(1) = 1+1 = 2`)

Yksinkertaisuutensa vuoksi kielessä ei ole mitään valmiita tietotyyppejä kuten totuusarvoja tai numeroja, mutta ne voidaan ilmaista sen avulla enemmän tai vähemmän monimutkaisesti. Totuusarvojen ilmaisu on hyvin yksinkertaista (Alonzo) Churchin booleaneilla:

- tosi = `λx.λy.x` eli valitaan ensimmäinen termi
- epätosi = `λx.λy.y` eli valitaan jälkimmäinen termi

Jos määritellään numero `0 = λx.λy.y` ja `1 = λx.λy.x y`, niin kaikki numerot voidaan esittää Churchin numeraaleilla rekursiivisesti:

`2 = λx.λy.x(x y)`

`3 = λx.λy.x(x(x y))` jne.

Eli suuret numerot esitetään pitkänä funktiokutsujonona. Churchin numeraalit ovat samalla esimerkki korkeamman asteen funktioista (higher-order function) – funktio ottaa argumentikseen funktion ja palauttaa funktion. Ne ovat hyvin tyypillisiä lambdakalkyylille ja funktionaaliselle ohjelmoinnille. [3, s. 32–34] [10, luku 5.2]

Koska lambdakalkyylissä ei ole Turingin koneen tapaan sisäistä tilaa, sillä ei voi ilmaista imperatiivisesta ohjelmoinnista tuttuja `for`- tai `while`-silmukoita. Toisto täytyy esittää funktionaalisessa ohjelmoinnissa rekursion avulla. Rekursiokutsussa parametrien nimet korvataan kutsuttavassa lausekkeessa kutsuhetken arvoilla eli funktio kutsuu itseään uusiksi, kunnes tietty ehto täyttyy. Rekursio voidaan ilmaista hajaantuvalla kombinaattorilla (divergent combinator) `omega = λf.(λx.f(x x)) (λx.f(x x))`. Hajaantumisella tarkoitetaan (call-by-value) tilannetta, jossa rekursiokutsu ei etene ikinä normaalimuotoon (normal form) eli konkreettiin paluuarvoon. Tällaisen tilanteen varalle on olemassa `omega`n yleisempi muoto ”kiintopiste-kombinaattori” (fixed-point combinator) `fix = λf.(λx.f(λy.x x y))(λy.f(λy. x x y))`.
[10, s. 69, 71]

Lambdakalkyyliin on myöhemmin tehty laajennuksina helpompaa syntaksia mm. numeroille, tuplille ja muille tietorakenteille. Laajennuksena on saatu tehtyä myös tuki mutatoituvuudelle ja poikkeustenkäsittelylle. Tässä työssä myöhemmin tarkasteltava Haskell-ohjelmointikieli perustuu juuri tällaiseen laajennettuun (tyypitettyyn) lambdakalkyyliin. [10, s. 58]

2.4 Tyypitetty lambdakalkyyli

Lambdakalkyylin matemaattinen tausta laajeni myöhemmin olennaisesti. Alonzo Church jatkoi lambdakalkyylin kehittämistä lisäämällä funktiomäritykselle tyypityksen; funktio esimerkiksi ottaa vastaan kokonaisluvun ja palauttaa totuusarvon. Hänellä oli tarkoituksena luoda tyypitetystä lambdakalkyylista malli matemaattiselle logiikalle. Ensimmäinen versio aiheesta pystyi esittämään propositiologiikkaa lambdakalkyylin termein. Jatkokehityksen myötä aiheet lähentyivät entisestään ja päädyttiin nk. Curry–Howard-vastaavuuteen, jonka perusteella matemaattisia todistuksia voi pitää ohjelmina ja väitteitä tyyppeinä.

Tietokoneohjelman ja matemaattisen logiikan välille oli siis saatu todistettua yhteys, ja näin ohjelma voidaan perustella muodollisesti oikeaksi. [11, luku 1] Tämä luo funktionaaliselle ohjelmoinnille valttikortin muihin verrattuna – sillä on vahva matemaattinen perusta, johon voi tukeutua kirjoitettaessa koodia, jonka on hyvin kriittistä toimia oikein. [6, osa 1, luku 2]

Ohjelmointikielen tyyppijärjestelmä auttaa ohjelmoijaa tekemään toimivia ohjelmia. Tyyppejä voi pitää joukkona arvoja, joita voi funktioilla (morfismeilla) kuvata toiseksi. Tavallinen esimerkki tyypistä on kokonaisluku (integer) tai merkkijono (string). Ohjelmakoodin käännösvaiheessa korkeamman tason kielestä konekieleen, kielen kääntäjä (compiler) tarkastaa koodin perinteisesti muun muassa sanastollisten (lexical) virheiden takia. Tyypitetyissä kielissä kääntäjä tarkistaa myös muuttujien tyyppien yhteensopivuuden. Mitä vahvempi kielen tyyppijärjestelmä on, sitä enemmän virheitä kääntäjä saa kiinni. Tyypityksen ansiosta vältetään ohjelmavirheitä, joissa funktioihin syötetään väärän tyyppisiä alkuarvoja. [6, osa 1, luku 2]

Jos kahden funktion palautus- ja lähtötyypit vastaavat toisiaan, niistä voi tehdä yhdistetyn funktion (funktiokomposition). [6, osa 1, luku 2] Funktiokompositiota helpottaa lambdakalkyylille tyypilliset puhtaat funktiot, jotka käyttäytyvät ennustettavasti [9, s. 5]. Funktiokompositiolle löytyy matemaattinen perusta abstraktin algebran osa-alueesta kategoriateoriasta. [6, osa 1, luku 1] Sen sisältämän muutaman yksinkertaisen säännön perusteella funktionaalisen kielen tyyppijärjestelmää voidaan yksinkertaistetusti pitää kategoriana. Näin kategoriateorian ideoita voidaan soveltaa funktionaaliseen ohjelmointiin.

2.5 Kategoriateoria

Kategoriateorian voidaan sanoa olleen hyvin kiinnostava aihe tietojenkäsittelytieteilijöiden keskuudessa 1990-luvun lopulla. Tutkimuksen voinee väittää huipentuneen vuoden 1991 Eugenio Moggin ”Notions of computation and monads” -artikkeliin. Siinä hän esittää idean siitä, miten kategoriateorian rakennetta monadi voidaan käyttää laskutoimitusten kuvaamisessa funktionaalisessa ohjelmoinnissa. Monadit ovat nykyaikaisessa Haskellissa tapa hallita luvussa 2.2 mainittuja sivuvaikutuksia. [6, osa 1, luku 2]

Kategoriateorian osaaminen ei missään nimessä ole välttämätöntä, jotta voisi ohjelmoida funktionaalisilla kielillä. [8, s. 353] Toisaalta varsinkin monadien voi hyvin väittää olevan melkoisen mielenkiinnon kohteena ohjelmoijien keskuudessa. Tästä kertoo lukuisat eri tutoriaalit, seminaaripuheet ja keskustelu sosiaalisessa mediassa. Kenties ensimmäinen lause, johon monadeista kiinnostunut henkilö törmää on se, että monadi on monoidi endofunktorien kategoriassa[1]. Kyseinen lause herättää varmasti enemmän kysymyksiä kuin vastauksia. Sen voi kuitenkin sanoa sisältävän kaikki tärkeimmät termit, mitä perusohjelmoijan on hyvä

[1] https://medium.com/@felix.kuehl/a-monad-is-just-a-monoid-in-the-category-of-endofunctors-lets-actually-unravel-this-f5d4b7dbe5d6

tietää. Edellä mainitut termit, sekä niihin liittyvä applikatiivi, esitellään tässä luvussa. Sitä ennen tarkastellaan hieman algebraa, joka on kaikkien näiden termien taustalla.

Algebra ja algebralliset rakenteet

Algebra on yksi matematiikan päähaaroista, jossa käytetään matemaattisia laskutoimituksia kuvaamaan erilaisten asioiden välisiä suhteita. Sille on tyypillistä asioiden abstrahoiminen – esimerkiksi alkeisalgebrassa tiettyjen numeroiden sijasta käytetäänkin jotain muuttujaa, joka voi kuvata mitä tahansa numeroa. Joukkoa asioita voi kuvata omiksi algebrallisiksi rakenteiksi, jos niiden välillä pätee tietyt (lasku)säännöt. Rakenteita on useita yksinkertaisista monimutkaisempiin. Yksinkertaisemmat rakenteet eivät noudata niin montaa sääntöä, kuin monimutkaisemmat. Rakenteet voivat toteuttaa jotkut tai kaikki seuraavista säännöistä, jotka liittyvät seuraaviin tilanteisiin: [12]

- Miten joukon alkiot käyttäytyvät yhteen- ja kertolaskun kanssa (onko laskutoimitusten järjestyksellä väliä - assosiatiivisuus)
- Tuottaako kahden saman rakenteen alkion laskutoimitus saman rakenteisen alkion eli onko se suljettu (vrt. miten kahden kokonaisluvun summa on aina kokonaisluku)
- Onko olemassa nolla-alkio tai identiteettialkio, jonka laskutoimitus alkion kanssa ei muuta sitä (vrt. miten 1 + 0 = 1 ja 1* 1 = 1)
- Onko olemassa käänteisalkio, jonka laskutoimitus jonkin alkion kanssa muuttaa sen nolla-alkioksi
- Toteuttaako rakenne yhden vai kaksi laskutoimitusta

Jotkut algebralliset rakenteet voi tunnistaa yleistyksenä tutuista rakenteista. Tällaisia rakenteita ovat mm. ryhmät, renkaat ja kunnat. Rengas yleistää kokonaislukujen yhteen- ja kertolaskun ja kunta yleistää mm. reaali- ja kompleksilukujen neljä laskutoimitusta. Ryhmä abstrahoi symmetrioita, ja sen voinee väittää olevan helpoimmin omaksuttava rakenne ja hyvin hyödyllinen poikkitieteellisesti. [12] Esimerkiksi hiukkasfysiikan standardimallin vuorovaikutusvoimat perustuvat tietynlaisiin symmetriaryhmiin. [13, s. 5] Tämä on samalla yksi hyvä lisäesimerkki sen tärkeydestä, että asioilla on vahva matemaattinen pohja.

Monoidi

Monoidi on muuten samanlainen rakenne kuin ryhmä, mutta siltä puuttuu käänteisalkio. Se on suljettu, assosiatiivinen ja sillä on identiteettialkio. Monet hyvin tavalliset tietorakenteet tai tyypit ovat monoideja. Näitä ovat mm. kokonaisluvut, merkkijonot ja listat. Listoja merkitään ohjelmointikielissä usein hakasulkeilla ” `[]` ”. Esimerkiksi kahden listan konkatenointi tuottaa listan, tyhjä lista `[]` on identiteettialkio ja listojen konkatenointi on assosiatiivinen (vrt. miten `[1]+[2]+[3] = [1,2]+[3] = [1]+[2,3] = [1,2,3]`). [3, s. 560–567]

Kategoria

Kategoria on periaatteessa yksinkertainen, mutta korkean abstraktiotason järjestelmä, joka koostuu joukoista ja niiden välisistä kuvauksista. Joukkoja (jotain kokoelmaa alkioita) ja niiden välisiä suhteita esitetään usein

graafina (verkkona), jossa joukoista on jokin polku toiseen joukkoon. Polut ovat tässä tapauksessa suunnattuja eli niitä pääsee kulkemaan vain tiettyyn suuntaan. Jotta tästä graafista voi tulla kategoria, sen pitää täyttää muutama sääntö algebrallisten rakenteiden tapaan. Useamman joukon välisistä poluista pitää voida tehdä uusi yhdistetty polku ja sen pitää kuvata alkuperäinen joukko lopullisekseen välittämättä siitä, missä järjestyksessä kaikkia välipolkuja kuljetaan – sen pitää olla assosiatiivinen. Huomataan, että kategorian määritelmä vaatii funktiokompositiota, joka on koko funktionaalisen ohjelmoinnin ydinteema. Jokaiselta joukolta pitää myös olla polku takaisin itseensä eli joukoilla on identiteettialkio. [14, s. 8] [6, osa 1, luku 1]

Funktori

Funktori on käsite, jolla tarkoitetaan kategoriateoriassa tapaa kuvata kategoria toiseksi. Koska funktionaalinen kieli koostuu vain yhdestä kategoriasta, puhutaan endofunktoreista, jotka kuvautuvat samaan kategoriaan. Käytännössä funktori on rakenne, joka voi sisältää alkioita, joihin jokaiseen pitää voida toteuttaa funktio (yleensä kutsuttu nimellä ”`fmap`”) niin, että sen rakenne pysyy samana. Käytännössä nämä rakenteet ovat ”säiliöitä” (container), kuten listat, tuplat ja puut. Esimerkiksi listaan `[1,2,3]` voi toteuttaa funktion, joka lisää jokaiseen alkioon 1, ja saadaan uusi lista `[2,3,4]`. [6, osa 1, luku 7]

Funktorin määritelmä sisältää muutaman hyvin olennaisen funktionaalisen ohjelmoinnin ominaisuuden. Funktorissa kutsutaan funktion `fmap` sisällä toista funktiota, jota sovelletaan jokaiseen alkioon. Kyseessä on siis korkeamman asteen funktio (ks. luku 2.3). Jotta `fmap` voi käydä jokaisen alkion läpi, sen pitää olla pohjimmiltaan rekursiivinen funktio, sillä lambdakalkyyli ei tue silmukoita. Viimeiseksi funktorit liittyvät vahvasti aiemmin mainittuun deklaratiiviseen ohjelmointitapaan. Sen sijaan, että esimerkiksi listaa iteroitaisiin `for`-silmukalla ja lisätään jokaisen listan alkioon jokin luku, niin sama voidaan funktionaalisella kielellä kuvata kompaktisti seuraavanlaisesti: `map (+1) [1,2,3]`. [3, s. 315–316]

Funktoreihin liittyy vielä yksi korkeamman tason käsite luonnollinen muunnos (natural transformation), joka on kuvaus funktorista toiseen (esim. listan muuttaminen puuksi). [6, osa 1, luku 10]

Applikatiivi

Applikatiivi on monoidinen funktori, jolla on mahdollista kohdistaa jonkin säiliön sisältämät funktiot toiseen säiliöön (esim. lista funktioita listaan arvoja). Sen ideana on, että useamman funktion kohdistaminen esim. listaan ei tuota useaa eri listaa tai listaa listan sisällä, vaan yhden listan. Tähän listaan on konkatenoitu kummankin funktion palauttamat arvot. Tämän takia applikatiivi on monoidinen – kahden listan konkatenointi tuottaa listan. [3, s. 650, 660] Asiaan liittyy termi ”litistys” (flattening). Monoidi litistää monimutkaisen rakenteen, kuten listan listoja ” `[[1,2], [3,4]]` ” yksinkertaisempaan muotoon, eli pelkäksi listaksi ” `[1,2,3,4]` ”. [3, s. 702]

Monadi

Monadilla on kategoriateoriassa useampikin määritelmä:

- monoidi endofunktorien kategoriassa
- applikatiivinen funktori
- triadi (T, η (eeta), μ (myy)) (huom. monadin nimen voi sanoa olevan monoid + triad = monad)

Viimeisessä T on endofunktori ja η ja μ ovat luonnollisia muunnoksia, joilla tarkoitettiin kuvausta funktorien välillä. η kuvaa endofunktorin T identiteettialkion itsekseen. μ kuvaa rakenteen, jossa T on T:n sisällä, pelkäksi T:ksi. Tämä vastaa applikatiivissa käsiteltyä tilannetta, jossa sisäkkäisestä listasta `[[]]` litistetään pelkkä lista `[]`. Monadi on siis monoidi endofunktorien kategoriassa, koska se sisältää funktorien välisiä kuvauksia, ja monoidien tavoin se litistää sisäkkäiset rakenteet yksinkertaisemmaksi. [6, osa 3, luku 15] Monadien käsittelyä ohjelmoinnin puolella jatketaan luvussa 3.6.

3 Funktionaalinen ohjelmointi ja Haskell

Kuten luvussa 2.1 mainittiin, funktionaalinen ohjelmointi perustuu lambdakalkyyliin. Siinä ei ole imperatiivisen ohjelmoinnin tapaan sisäistä tilaa, vaan kaikki muuttujat ovat mutatoitumattomia. Lambdakalkyylille ominaiset puhtaat funktiot aiheuttavat päänvaivaa, kun tehdään ohjelmia, jotka vuorovaikuttavat käyttäjän tai ulkopuolisen datan kanssa (luku 2.2). Tätä varten kappaleessa 2.5 esiteltiin kategoriateoriaa ja erityisesti monadeja, joilla nykyaikaisessa Haskellissa toteutetaan sivuvaikutuksia.

Funktionaalisessa ohjelmoinnissa ohjelma rakennetaan yhdistämällä puhtaita funktioita toisiinsa. Puhtaat funktiot palauttavat saman arvon joka kerta, kun niitä kutsutaan samoilla parametreilla – ne eivät aiheuta sivuvaikutuksia (luku 2.2). Kieli, kuten Haskell, on puhtaasti funktionaalinen, jos se toimii juuri tällä tavalla. On olemassa myös kieliä, kuten F# (F Sharp), jotka yhdistävät funktionaalisia ja olio-ohjelmoinnin ajattelumalleja. [15] Keskeinen ajattelumalli funktionaalisessa ohjelmoinnissa on funktioiden pitäminen ns. ensimmäisen luokan kansalaisina. Niitä voi käsitellä, kuin mitä tahansa muuta dataa ohjelmassa: niitä voi antaa muiden funktioiden argumentiksi, palauttaa funktiosta ja asettaa muuttujaan. [3, s. 30]

Tässä luvussa esitellään Haskellia ja paneudutaan puhtaan funktionaalisen ohjelmoinnin erikoisominaisuuksiin, jotka pohjimmiltaan periytyvät lambdakalkyylista ja kategoriateoriasta.

3.1 Haskell

Haskell on yleiskäyttöinen, korkean tason puhtaasti funktionaalinen ohjelmointikieli. Haskellin ensimmäinen versio ilmestyi vuonna 1990. Sen nykyisessä logossa näkyy nerokkaalla tavalla sen tausta lambdakalkyylissa ja kategoriateoriassa: siinä on yhdistetty monadien bind-operaattori ”>>=” ja lambdakalkyylin lambda-kirjain. Yleisimpien oliokielten, kuten Pythonin ja JavaScriptin, tavoin sitä voi käyttää laajamittaisesti eri sovelluksiin ja se tarjoaa vahvan abstraktion tietokonearkkitehtuurin yksityiskohdista. Abstraktion vuoksi ohjelmoijan ei mm. tarvitse itse huolehtia tietokoneen muistin käytöstä (roskien keruu) ja kielen mukana tulee valmiiksi tuki yleisille tietorakenteille. Haskell saa nimensä loogikko Haskell Currysta (1900–1982), jonka nimi on epäsuorasti mainittu jo kappaleen 2.4 Curry–Howard-vastaavuudessa. Hänen vaikutuksensa funktionaalisen ohjelmoinnin kehityksessä näkyy myös termissä ”currying”, jota käsitellään edempänä. [16]

Tyyppijärjestelmä

Haskellilla on vahva staattinen tyypitys ja sen tyyppijärjestelmä on useimpiin muihin kieliin verrattuna vahvempi ja ilmaisuvoimaisempi. Haskell tukee parametrista polymorfismia (vastaa mm. Javan geneerisyyttä) ja se on ensimmäisenä tuonut tyyppiluokat, joista muutama (funktori, monadi) mainittiin luvussa 2.5. Tyyppiluokkia voi pitää vahvempana versiona esim. Javan rajapintaluokista ja ne mahdollistavat korkean tason abstraktiota ja tilapäistä polymorfismia (ad-hoc polymorphism) [9, s. 134]

Vahvan staattisen tyypityksen seurauksena Haskellin kääntäjä (Glasgow Haskell Compiler, GHC) pystyy huomaamaan huomattavan määrän tyyppivirheitä käännösaikana. Haskell-ohjelmoijien keskuudessa kiertää usein sanonta ”jos se kääntyy, se toimii”, merkiten vahvaa luottoa kääntäjän toimivuuteen. [9, s. 6] Vertailuna Python ja JavaScript käyttävät heikkoa dynaamista (ajonaikaista) tyypitystä [17] [18], joka voi johtaa ohjelman ajonaikaiseen kaatumiseen [19]. GHC tukee myös tyyppipäättelyä, eli funktion argumenttien tyyppejä ei välttämättä tarvitse erikseen kirjoittaa. Tyyppien kirjoittamista voi kuitenkin pitää hyvänä ohjelmointitapana. [6, osa 1: luku 2]

Syntaksi

Tarkastellaan Haskellin syntaksia pikalajittelualgoritmin (quicksort) kautta. Seuraava koodi on ehkä kompaktein esitys kyseiselle algoritmille (vasemmassa reunassa rivimerkinnät viittausten helpottamiseksi) [20].

```
1 quicksort :: Ord a => [a] -> [a]
2 quicksort []      = []
3 quicksort(x:xs) =
4     quicksort smaller ++ [x] ++ quicksort larger
5     where
6         smaller  = [a | a <- xs, a <= x]
7               larger    = [b | b <- xs, b > x]
```

Rivillä 1 määritetään funktiolle nimi `quicksort`, ja määritetään sen tyyppi. ”`Ord a =>`”on tyyppiluokan signatuuri. Sillä täsmennetään, että funktio toimii vain tyypeille, joita voidaan jotenkin järjestää keskenään. ”`[a] -> [a]`” tarkoittaa, että funktio ottaa vastaan listan alkioita, ja palauttaa saman tyyppisen listan.

Pikalajittelu-esimerkissä käytetään rekursiota. Sen voisi ohjelmoida myös silmukoilla iteratiivisesti (`for`, `while`), mutta lambdakalkyyli ei tue niitä (luku 2.3). Täten Haskellissa algoritmi täytyy esittää rekursiivisesti. Rivillä 2 määritetään rekursion päättymishaara: jos lista on tyhjä (`[]`), palautetaan tyhjä lista. Riveillä 3–7 määritetään rekursiohaara siinä tilanteessa, kun lista ei ole tyhjä. Tyhjän ja ei-tyhjän listan vertailussa käytetään ns. hahmon sovitusta (pattern matching) ”`x:xs`”. Hahmon sovitus on Haskellissa hyvin yleistä, ja sillä voidaan kompaktisti esittää `if-else`-tyyppisiä tilanteita [3, s. 231]. Lajittelu tehdään niin, että määritetään listat `smaller` ja `larger`, jotka riippuvat käsittelyssä olevasta listan alkiosta. Kyseiset listat luodaan listakoosteella (list comprehension) ja käsitellään rekursiivisesti kutsumalla `quicksort`-funktiota uudestaan. Lopulta lasketut listan osat konkatenoidaan yhteen (`++`).

Suorituskyky

Korkean (abstraktio)tason kielet ovat yleensä matalan tason kieliä, kuten C-kieltä, hitaampia suorituskyvyltään. Esimerkiksi edellä käsitelty pikalajittelualgoritmi on Haskellissa paljon kompaktimpi kuin C:ssä, mutta oikealla ja nerokkaalla toteutuksella C-kielessä se on paljon nopeampi, eikä vie lainkaan ylimääräistä muistitilaa. Haskell vie ylipäänsä paljon muistia pääosin mutatoitumattomuuden, osittain laiskan suorituksen takia. Toisaalta varsinkin nykyaikana voisi väittää, että harva laite (poikkeuksena kenties sulautetut järjestelmät) vaatii nimenomaan suorituskyvyltään tehokkaita ohjelmia. [16]

3.2 Laiska suoritustapa

Haskell käyttää useimpiin muihin kieliin verrattuna erilaista ohjelman suoritustapaa. Termi liittyy siihen, miten ja millä säännöillä ohjelmakoodia suoritetaan. Säännöt liittyvät muutamiin ohjelmointikielien metatason ratkaisuihin, jotka ovat syystä tai toisesta nähty järkeväksi ottaa käyttöön. Keskeisiä kysymyksiä asiassa ovat, miten määritetään lopullinen arvo funktiolle syötettävälle parametrille, lasketaanko jokainen funktion parametri ja missä järjestyksessä ne mahdollisesti lasketaan. Keskitytään kysymykseen siitä, lasketaanko jokainen funktion parametri. Jos ne lasketaan aina ennen funktion suoritusta, puhutaan ahkerasta kielestä (strict tai eager evaluation). Tavallisimmat kielet, kuten tässä työssä vertailukohteena olevat Python ja JavaScript, ovat ahkeria kieliä. Haskell sen sijaan on laiska kieli (non-strict tai lazy evaluation), eli funktion parametrit lasketaan vasta sitten, kun niitä tarvitaan. Tämä voi hyvässä tapauksessa nostaa ohjelman tehokkuutta, jos voi välttää turhien laskutoimitusten suorittamista. Mahdollisesti suoritettavia arvoja varten Haskell tallentaa muistiin ”ajatuksen” (thunk) siitä, miten arvo voidaan laskea [9, s. 163–170]

Funktionaalisen ohjelmoinnin kehityshistoriassa 1970- ja 1980-luvuilla laiskuuden toteutus herätti innostusta Von-Neumannista poikkeavien tietokonearkkitehtuurien kehittämiseen (luku 2.2). Tällaisia olivat ”tietovirta-arkkitehtuuri” (dataflow architecture) ja ”verkkosievennyskone” (graph reduction machine). Niiden kehitys johti kuitenkin suurimmaksi osaksi umpikujaan. Myöhemmin havaittiin, että erikoistunut arkkitehtuuri ei tuonutkaan tehoetua; ohjelman optimoitu kääntäminen Von-Neumanniin oli tehokkaampaa. [21, s. 2]

Laiskuus mahdollistaa ohjelmoinnin äärettömien ja syklisten tietorakenteiden kanssa. Tyypillisin esimerkki tästä on äärettömän pitkien listojen käyttö. Laiskuus auttaa osaltaan pitämään Haskellin syntaksia sille tyypillisesti hyvin kompaktina. Esimerkkinä tästä on mm. listakoosteet (ks. luku 4.1). Laiskuuden takia äärettömiä ja äärellisiä tietorakenteita voi käyttää huoletta keskenään, joka ahkerissa kielissä aiheuttaisi ylimääräistä päänvaivaa [8, s.67].

Laiskuuden haittapuolena on tehokkuus – vaikka laiska suoritus voi sopivassa tapauksessa nopeuttaa suoritusta, se ei keskimäärin vähennä suoritettavien laskelmien määrää. Jos arvoja joudutaan laskemaan thunkeista, se on huomattavasti hitaampaa kuin niiden normaali laskenta. Thunkit kuluttavat tämän lisäksi

paljon muistia ja sen käyttöä voi olla hankala etukäteen tietää. Arvaamaton muistinkäyttö voi johtaa muistivuotoihin, ja tätä varten Haskelliin on myöhemmin tuotu myös mahdollisuus suorittaa haluttu ohjelmakoodi ahkerasti. [21, luku 3] [9, luku 5] Laiskuuden takia on vaikea tietää tarkkaan, milloin jokin arvo oikeasti lasketaan. Sen takia tavallisia sivuvaikutteisia ohjelmia on lähes mahdotonta suorittaa luotettavasti laiskana. Haskellin historiaa käsittelevässä artikkelissa [21] perustellaan ahkeran kielen käytön johtavan välttämättömästi sivuvaikutusten käyttöön. Tämän takia Haskell oli etukäteen päätetty laiskaksi kieleksi, vaikka sen puutteet olivat tiedossa. [21, luku 3]

3.3 Currying

Yleisemmissä ohjelmointikielissä funktiot voivat samaan aikaan ottaa vastaan useamman argumentin. Lambdakalkyylissa ja Haskellissa funktioiden on kuitenkin mahdollista ottaa kerralla vastaan vain yksi argumentti, ja palauttaa yksi arvo. Useamman argumentin funktiot esitetään lambdakalkyylissa sisäkkäisinä (korkeamman asteen) funktioina, joissa ulompi funktio palauttaa aina uuden funktion seuraavan argumentin käyttöön (luku 2.3). Tästä hieman teknisen oloisesta suoritustavasta käytetään nimeä currying. Asian voi paremmin nähdä esimerkkifunktiosta, joka ottaa vastaan kaksi kokonaislukua ja palauttaa niiden summan:

```
1 summaa :: Integer -> Integer -> Integer
2 summaa a b = a + b
```

Funktio näyttäisi rivillä 1 ottavan vastaan kaksi kokonaislukuargumenttia ja palauttavan uuden kokonaisluvun. Haskellin tyyppikonstruktorin, eli nuolioperaattorin ominaisuuksien takia rivi 1 tarkoittaa samaa kuin `Integer -> (Integer -> Integer)`. Tämä luetaan niin, että funktio ottaa vastaan kokonaisluvun, ja palauttaa suluissa määritetyn funktion. [3, s. 144–146]

3.4 Osittainen suoritus

Currying mahdollistaa käytännön ohjelmointiin liittyvän funktion osittaisen suorittamisen (partial application). Siinä funktiolle ei anneta kaikkia sen argumentteja. Muissa ohjelmointikielissä tämä todennäköisesti aiheuttaisi virheen, mutta Haskellissa se on oletusarvoisesti mahdollista curryingin takia. Jos esimerkiksi aiemmin käsitellylle `summaa`-funktiolle antaisi argumentiksi vain ”`summaa 1`”, se vain palauttaisi uuden funktion, joka odottaa toista argumenttia. Tätä palautettua funktiota kutsutaan sulkeumaksi (closure) [9, s. 74]. Sulkeumassa on tällöin yksi argumentti vähemmän, eli sen paikkaisuus (arity) pienenee. Osittaisella suorittamisella voi hyväksikäyttää aiemmin määriteltyjä funktioita. Jos määritetään esimerkiksi funktio `summaaYksi = summaa 1`, niin ”`summaaYksi 1`” palauttaa luvun 2. Osittaisesti suoritetut funktiot ovat näppäriä varsinkin käytettäessä niitä listojen `map`-funktioiden kanssa. `Map` on hyvin yleisesti käytetty rekursiivinen korkeamman asteen funktio, joka soveltaa annettua funktiota jokaiseen annetun listan

alkioon [3, s. 315]. Tällöin `summaaYksi`-funktiota voi käyttää `map`in kanssa seuraavasti: `map (summaaYksi) [1,2,3]`, joka palauttaa listan `[2,3,4]`. [3, s. 144–146]

3.5 Anonyymit funktiot

Hieman vastaavalla tavalla, kuin edellisessä luvussa mainitut funktion osittaiset suoritukset, Haskellin anonyymit funktiot mahdollistavat näppärämmän tavan käyttää korkeamman asteen funktioita. Usein esimerkiksi listojen `map`-funktiota käyttäessä törmää ongelmaan, että sen sisällä käytettävä funktio pitäisi ensin määritellä. Varsinkin yksinkertaisten funktioiden, kuten seuraavan esimerkin mukaisen luvun kolminkertaistamisen, erikseen määrittelystä tulee nopeasti vaivalloista:

```
triple :: Integer -> Integer
triple x = x * 3
```

Edellisen koodin voi ilmaista anonyymin funktion kanssa seuraavalla tavalla paljon näppärämmin suoraan `map`-funktion sisällä:

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

Edellisessä koodissa ”\” on Haskellin tapa kirjoittaa luvusta 2.3 tuttu λ-kirjain. `\x -> x + 1` vastaa käytännössä suoraan lambdakalkyylin ” `λx.x+1`” lambdafunktiota. Tämän takia anonyymeja funktioita kutsutaan myös muun muassa lambdafunktioiksi ja funktioabstraktioiksi. Anonyymien funktioiden käyttö aiheuttaa muutaman rajoituksen niiden käytössä. Koska niillä ei ole nimeä, ne eivät voi kutsua itseään rekursiivisesti. Ne rajoittavat myös aiemmin mainitun hahmon sovituksen käytön vain yhteen tapaukseen. Useamman hahmon sovittamisessa pitää käyttää vaivalloisempaa `case`-syntaksia. [9, s. 70–71]

3.6 Sivuvaikutusten hallinta

Sivuvaikutukset (luku 2.2) käsitellään Haskellissa luvussa 2.5 esitetyillä monadeilla. Kategoriateorian näkökulmasta monadi on applikatiivinen funktori. Tälle abstraktille matemaattiselle rakenteelle on Haskellissa tehty käytännön toteutus tyyppiluokassa `Monad`:

```
class Monad m where                    //   m = T kategoriateoriassa
(>>=) :: m a -> (a -> m b) -> m b      //   = μ
return ::     a            -> m a      //   = η
```

`Monad`-tyyppiluokan pitää lisäksi noudattaa monoidien (luku 2.5) tapaan identiteettisääntöä ja assosiatiivisuutta. Identtisyys pitää olla molemminpuoleinen:

```
m >>= return = m
```

```
return x >>= f = f x
```

`Return` käyttäytyy siis identiteettialkion tavoin, eikä muuta laskutoimitusta mitenkään. Se vain käärii arvon monadin sisälle. Seuraava koodi ilmaisee monadien assosiatiivisuutta; funktioiden järjestyksen vaihtamisella ei ole väliä lopputuloksen kannalta.

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

`Monad`-tyyppiluokan ”`return`” käärii puhtaan arvon monadin sisälle. `Bind`iksi kutsuttu ”`>>=`” funktio on monadien valttikortti. Sillä on mahdollista samaan aikaan ketjuttaa funktiokutsuja, ja litistää (luku 2.5) niiden tulos yksinkertaisemmaksi. Ketjutus on niin yleistä, että sille on tehty syntaktisena sokerina imperatiivisesta ohjelmoinnista tuttu `do`-notaatio. [3, s. 698–706] Seuraavassa esimerkki `bind`in käytöstä (1) ja `do`-notaatiosta (2), jotka vastaavat toisiaan:

(1)

```
f >>= \a ->
  g >>= \b ->
    h >>= \c ->
      return (a, b, c)
```

(2)

```
do
   a <- f
   b <- g
   c <- h
   return (a, b, c)
```

[8, s. 32]

Monadien toteutuksia on useita erilaisia eri käyttötarkoituksiin. Yksinkertaisin niistä on `Maybe`-monadi (käsitellään luvussa 4.4), jolla voi käsitellä mahdollisia ohjelman kaatumistilanteita. Toiseksi yksinkertaisimmalla `List`-monadilla voi kerätä tietoa kokoelmaksi. `IO`-monadilla voi käsitellä vuorovaikutusta (I/O, input/output) tietokoneen käyttäjän kanssa, eli käsitellä sivuvaikutuksia. Monadin ideassa siis yhdistyy kolme perustavanlaatuista ohjelmointikäsitettä yhdeksi, korkeamman tason käsitteeksi. Tämä korkea abstraktiotaso tekee siitä monikäyttöisen, mutta samalla vaikeasti ymmärrettävän. [8, s. 33–36]

Sivuvaikutusten käsittely monadeilla on viittauksellisesti läpinäkyvä. Kyseisellä termillä tarkoitetaan sitä, että lauseke voidaan korvata sen arvolla niin, että ohjelman toiminta ei muutu. Jos funktio voidaan korvata suoraan sen palauttamalla arvolla, se ei voi esimerkiksi muuttaa jonkun globaalin muuttujan arvoa. Funktionaalisessa ohjelmoinnissa termiä käytetään usein synonyymina puhtaalle funktiolle, vaikka ne

tarkoittavat hieman eri asiaa. [3, s. 1068–1070] Asiaa ei auta yhtään se, että eri lähteissä viittauksellisella läpinäkyvyydellä tarkoitetaan myös usein hieman eri asiaa.

Monadien kanssa yleinen `do`-notaation käyttö vaikeuttaa havaitsemaan, kuinka monadien laskutoimitukset suoritetaan sisäkkäisinä lambdafunktioina. Esimerkiksi usea peräkkäinen vuorovaikutus käyttäjän kanssa suoritetaan niin, että ensimmäinen toiminto ketjutetaan anonyymille lambdafunktiolle, joka ketjuttaa (bindaa) sen seuraavalle ja niin edelleen (edellisen sivun koodin (1) mukaisesti). Jokaisessa vaiheessa lambdafunktio saa ohjelman tilasta riippumattoman arvon, jota ei missään välissä ole tallennettu mihinkään globaaliin muuttujaan. Näin monadi pysyy muusta ohjelmasta irrallisessa kontekstissa ja se on viittauksellisesti läpinäkyvä. [20, luku 12]

Haskellissa on ollut monadeja ennen muitakin tapoja käsitellä sivuvaikutuksia, kuten ”dialogit”. Niiden käyttö oli virheherkkää eivätkä tukeneet bindin tapaan funktiokompositiota. [22, s. 4–5] Funktiokompositio on funktionaalisen ohjelmoinnin pääideoita ja monadit tukevat sitä. Käytännössä kuitenkin useiden eri vaikutusten yhdistäminen eri monadeilla samaan aikaan on hankalaa, ja sitä varten on olemassa edistyneempää osaamista vaativia monadien muuntimia (monad transformers) [3, s. 901].

3.7 Funktionaalisen ohjelmoinnin edut

Miksi funktionaalisesta ohjelmoinnista pitäisi ylipäätään välittää? Olio-ohjelmointiin verrattuna sillä on monia etuja. Funktionaalisen ohjelmoinnin tärkein etu liittyy varmasti puhtaisiin funktioihin ja sivuvaikutusten hallitsemiseen. Puhtailla funktioilla tehty ohjelma on huomattavasti helpompaa perustella oikein toimivaksi, ja vältytään mitä erikoisimmilta ohjelmabugeilta. Deklaratiivinen ohjelmakoodi voi auttaa helpommin ymmärtämään, mitä koodin on tarkoitus tehdä. Korkeampi abstraktiotaso helpottaa koodin uudelleenkäyttöä ja tekee ohjelmakoodista kompaktimman. Seuraavat edut voivat olla se valttikortti, jonka takia funktionaalinen ohjelmointitapa voisi yleistyä tulevaisuudessa. Ne liittyvät nykyään yhä ajankohtaisempiin asioihin: samanaikaisuuteen (concurrency), rinnakkaisuuteen (parallelism) ja massiivisten sovellusten hallintaan. [9, s. 18] [6, preface]

Samanaikaisuus liittyy muun muassa siihen, kuinka web-palvelin vastaa useaan samaan aikaan tulleeseen pyyntöön. Pyynnöt liittyvät usein pääsyyn johonkin jaettuun resurssiin, joka voi mahdollisesti muuttua pyyntöjen välissä. Samanaikaisuuden hallinta olio-ohjelmoinnilla on surullisenkuuluisa bugisuudestaan. [6, preface] Haskell tarjoaa samanaikaisuuden käsittelyyn ”Software Transactional Memory”-tavan, jota käytetään myös tietokannoissa. [9, s. 270]

Rinnakkaisuus liittyy koodin suorittamiseen useammalla tietokoneytimellä yhdellä kertaa ja sillä voi saada suuren tehokkuusedun. Funktionaalisen ohjelmoinnin matemaattisesta taustasta ilmenee montakin syytä, miksi rinnakkaisuus on hyvin luonnollista Haskellille. Lambdakalkyylin Church-Rosser-teoreeman mukaan lausekkeen sievennyksen järjestyksellä ei ole väliä lopputuloksen kannalta. [2, s. 6] Kategoriateoriassa kategorian määritelmä vaatii käytännössä samaa kuin edellä mainittu teoreema eli funktiokomposition assosiatiivisuutta (luku 2.5). Mutatoitumattomuuden takia rinnakkaislaskenta ei edes voisi sotkea jotain yhteistä tilaa.

Ohjelmistot käyvät nykyään yhä isommiksi ja monimutkaisiksi ja niiden hallinta imperatiivisella ohjelmointityylillä sivuvaikutuksineen on haastavaa. Sovellusten kasvaessa sivuvaikutuksia kasvaa sivuvaikutusten päälle, mikä helposti aiheuttaa arveluttavia bugeja. Puhtaat funktiot skaalautuvat paljon paremmin isoissa sovelluksissa. Lisäksi Haskellin vahva tyyppijärjestelmä auttaa osaltaan skaalautuvuutta havaitsemalla bugeja jo käännösvaiheessa, eikä tarvitse toivoa testien ottavan niitä myöhemmin kiinni. [9, s.18] [6, preface]

4 Funktionaalisuus yleisemmissä kielissä

Tällä hetkellä vallitsevassa ohjelmoinnin ajattelutavassa olio-ohjelmoinnissa (object-oriented programming) muodostetaan joukosta loogisesti yhteenkuuluvaa tietoa ja toiminnallisuutta tietorakenteita, joita kutsutaan olioiksi (object). Olioilla on jokin tila, joka voi muuttua ja ne voivat lähettää tietoa itsestään muille olioille. Ohjelmakokonaisuus saadaan toimimaan niiden yhteistoimintana. [23] Kyselytutkimuksen [1] mukaan yleisimmät ohjelmointikielet ohjelmistokehittäjien keskuudessa ovat olio-ohjelmointikieliä, jotka tukevat myös muita ajattelumalleja. Näitä kutsutaan myös multiparadigmakieliksi. Niihin kuuluvat mm. JavaScript/TypeScript, Python, Java, C# ja C++. Suosituinta puhtaasti funktionaalista kieltä Haskellia käyttää samaisen kyselyn mukaan vain 2,09 % kehittäjistä. [1]

Funktionaalisia ohjelmointitapoja, joita käsiteltiin luvussa 3, on haluttu ottaa mukaan edellä mainittuihin kieliin joko alusta asti, tai myöhemmin laajennuksena. Multiparadigmakielet ylipäänsä mahdollistavat vaihtoehtoisia toteutustapoja koodaamiselle. Esimerkiksi ison sovelluksen eri kokonaisuudet voivat olla koodattu eri tavalla. Ehkä on koettu tarkoituksenmukaiseksi ohjelmoida käyttöliittymä oliotyylillä, ja prosessointilogiikka proseduraalisesti tai funktionaalisesti? Yleisimmin käytetyt kielet varmasti hyötyvät tästä joustavasta ohjelmointitavasta, jossa ohjelmoijaa ei pakoteta tekemään asioita esimerkiksi ainoastaan funktionaalisella tai oliotyylillä. Joustavuuden varjopuolena on se, että eri ohjelmointiparadigmojen toteutus jää helposti hieman tyngäksi. Esimerkiksi funktionaalinen ohjelmointityyli Pythonilla näyttää päällisin puolin oikealta, Haskellin tapaiselta, mutta sisäisessä toteutuksessaan se käyttää ei-funktionaalisia keinoja. [23]

Python ja JavaScript ovat jo alusta alkaen suunniteltu multiparadigmakieliksi, ja ne tukevat päällisin puolin funktionaalista ohjelmointia. [23][24] Viime vuosina on koettu tarkoituksenmukaisena käyttää funktionaalista ohjelmointitapaa varsinkin web-ohjelmoinnin parissa JavaScriptillä. Sitä varten on kehitetty entistä enemmän funktionaaliseen ohjelmointiin nojaavia kirjastoja. Näitä ovat muun muassa JavaScript-kirjastot React ja Vue, jotka ovat julkaistu vuonna 2013 ja 2014. [25][26]

Tässä luvussa tarkastellaan, mitä ja miten luvussa 3 esitettyjä funktionaalisia ominaisuuksia yleisemmissä kielissä käytetään. Ominaisuuksien käyttöä vertaillaan Haskellin ”alkuperäiseen” toteutustapaan. Vertailussa kiinnitetään huomiota siihen, kuinka onnistuneesti funktionaaliset ominaisuudet ja ideat on saatu tuotua niiden pariin. Vertailukielinä käytetään yleisimpiä ohjelmointikieliä Pythonia ja JavaScriptiä.

4.1 Listakooste

Pythonissa on mahdollista käyttää listakoostetta (luku 3.1) (list comprehension) alustamaan ja täyttämään lista jonkin ehdon mukaisesti. Kyseistä ominaisuutta ei JavaScriptistä löydy. Tavallaan yksinkertaisin tapa listan luomiseen ja sen täyttämiseen on luoda tyhjä lista, ja lisätä sinne alkioita esimerkiksi for-loopissa:

```
1 squares = []
2 for i in range(10):
3     squares.append(i * i) // squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Saman asian voi koodata yhdellä rivillä listakoostella seuraavasti:

```
squares = [i * i for i in range (10)]
```

Haskellissa sama koodi on seuraavanlainen:

```
squares = [i * i | i <- [0..9]]
```

Pythonin listakooste on Haskellin tapaan hyvin kompakti koodaustapa. Erona niissä on se, että Python ei käytä laiskaa suoritusta. Tämä rajoittaa sen käyttömahdollisuuksia – äärettömän tai todella ison listan luominen kaataa tai jumittaa tietokoneen. [23]

4.2 Laiskuus ja iterointi

Pythonissa ja JavaScriptissa on kummassakin tärkeä ominaisuus, joka antaa perustan funktionaalisen ohjelmoinnin käyttämiseksi: iteraattorit. Iteraattori on olio, joka esittää jotain tietorakennetta (säiliötä), kuten listaa asioita. Kummassakin kielessä on samankaltainen `iterator`-luokka tai -prototyyppi, jonka tulee toteuttaa `next()`-metodi, joka palauttaa tietorakenteesta uuden alkion yksi kerrallaan. Tärkeimpänä asiana iteraattoreissa on se, että ne palauttavat uuden alkion vain tarvittaessa, eli laiskalla suoritustavalla (luku 3.2). Kumpikin kieli on oletusarvoisesti ahkera, joten iteraattorien käyttö vaatii tavallaan ylimääräisen toimenpiteen näiltä kieliltä.

Iteroituvuutta tukevat tietorakenteet ovat tyyppiä `iterable`. Yleisimmät tietorakenteet, kuten listat, ovat oletusarvoisesti sen tyyppisiä. Funktionaalisesta ohjelmoinnista tuttujen rekursiivisten funktioiden, kuten `map` ja `filter`, käyttö Pythonissa ja JavaScriptissa vaatii, että iteroitava tietorakenne on `iterable`. `For`-silmukoiden syntaktisena sokerina pidettävät `for… in` (python) ja `for…of` (JavaScript) silmukat perustuvat myös iterableihin. Ne ovat näppärämpi keino iteroida esimerkiksi lista läpi niin, ettei tarvitse itse imperatiiviseen tyyliin kirjoittaa `for`-silmukan toteutusta. Seuraavassa esimerkki `for…in` käytöstä Pythonilla:

```
lista = [1, 2, 3]
for i in lista:
    print(i)
```
[27] [28]

Muita esimerkkejä iterableja hyödyntävistä ohjelmointitavoista kummassakin kielessä ovat `spread`-syntaksi ja destrukturointi-asetus. Ne ovat tavallaan päinvastaisia operaatioita. Spread-syntaksilla (…) voi ”levittää” esimerkiksi listan yksittäisiksi alkioikseen. Destrukturoinnilla (”purkamisella”) voi kerätä yksittäisiä alkioita muuttujaan. Kumpaakin ominaisuutta voi käyttää moneen tarkoitusperään syntaktisena sokerina. Spread-syntaksia voi käyttää esimerkiksi muuttamaan lista yksittäisiksi alkioikseen funktiokutsun sisällä. Seuraavassa koodiesimerkki JavaScriptilla:

```
function sum(x, y, z) {
  return x + y + z;
}
const numbers = [1, 2, 3];
console.log(sum(…numbers));
```
//tulostaa 6 [29]

Destrukturointi-asetusta, joskus myös nimellä rest-syntaksi (`…rest`), voi käyttää esimerkiksi seuraavasti JavaScriptilla:

```
let [a, b, …rest] = [1, 2, 3, 4, 5];
console.log(rest);
```
// tulostaa Array [3, 4, 5]

Destrukturointia voi käyttää myös ilman rest-syntaksia, jolloin sillä voi purkaa olioliteraalin attribuutit uuteen muuttujaan:

```
const user = { //olioiteraali
   id: 1
   name: foo
};

const { id, name } = user;
```
//destrukturointi

Destrukturoidun olion attribuutteihin voi nyt viitata suoraan. Normaalisti esimerkiksi attribuuttiin ”`id`” pitäisi viitata ”`user.id`”. [30]

4.3 Anonyymit funktiot

Pythonissa, JavaScriptissa ja lukuisissa muissa kielissä voi käyttää Haskellista tuttua anonyymia funktiota. Haskellissa anonyymi funktio ilmaistiin kenoviivalla ”\” nuolella ”->” (luku 3.5). Pythonin ja JavaScriptin anonyymien funktioiden syntaksi eroaa aika paljon toisistaan. Anonyymeja funktioita on näppärää käyttää varsinkin edellisessä luvussa mainittujen `map`- ja `filter`-funktioiden kanssa, jotka tarvitsevat parametrikseen toisen funktion. Tarkastellaan, miten kummassakin kielessä lisätään listaan `lista = [1, 2, 3]` luku 1 mapilla anonyymin funktion avulla:

Python: `list(map(lambda n: n +1, lista))`

JavaScript: `lista.map(n => n +1)`

Pythonin ”`lambda n: n+1`” on paljon kömpelömpi tapa määrittää anonyymi funktio, kuin JavaScriptin nuolifunktio (arrow function). Nuolifunktio ”`n => n + 1`” on eittämättä hyvin näppärä, ja samankaltainen Haskellin vastaavan anonyymin funktion ”`\n -> n +1`” kanssa. Anonyymien funktioiden käytön rajoitteita käsiteltiin luvussa 3.5, ja vastaavat rajoitteet koskevat myös Pythonia ja JavaScriptia. [23] [31]

4.4 Monadit

Monadit ovat jokseenkin kuuma puheenaihe Internetissä ohjelmoijien keskuudessa. Usein monadeihin tutustuu ensimmäisen kerran web-ohjelmoinnin parissa asynkronisten (”eriaikaisten”) palvelupyyntöfunktioiden kautta. Ne ovat korkeamman asteen funktioita, jotka saavat parametrikseen palvelupyyntöön liittyvää tietoa, sekä takaisinkutsufunktion (callback function). Takaisinkutsufunktiota kutsutaan, kun palvelupyyntö on käsitelty. Pyynnön käsittely saattaa viedä paljonkin aikaa, ja ohjelman jumittumisen välttämiseksi tapahtuma suoritetaan omassa säikeessään. Johtavassa web-ohjelmointikielessä JavaScriptissä tätä varten on olemassa esimerkiksi Web Workers API. [32][33]

Asynkroniset palvelupyynnöt voivat myös epäonnistua, ja niitä seuraavat poikkeukset pitäisi käsitellä. Käytännössä palvelupyynnöissä on myös usein sisäkkäisiä palvelupyyntöjä, joita kutsutaan järjestyksessä. Jokaista seuraavaa palvelupyyntöä ennen pitää tarkistaa, epäonnistuiko edellinen pyyntö. Näin päästään jo muutaman useamman sisäkkäisen palvelupyynnön tapauksessa kömpelöön ja vaikeasti luettavaan koodiin, joka sisältää suurimmaksi osaksi poikkeustenkäsittelyä. Tätä kutsutaan usein ”callback-helvetiksi” ja sisenevien `if-else` – blokkien takia ”tuomion pyramidiksi”. [32]

Nykyaikainen ja parempi tapa käsitellä asynkronisia funktiokutsuja tapahtuu promisejen avulla. ”`Promise`” on sananmukaisesti lupaus siitä, että palvelupyynnön käsittelyn jälkeen siihen vastataan. Asynkronisen funktion kutsun jälkeen `promise` on ensin odottamassa käsittelyä tilassa ”`pending`”, ja sen arvo ei ole tiedossa (`undefined`). Jos käsittely onnistuu, tila muuttuu suoritetuksi (`fulfilled`), ja kutsutaan funktiota

”resolve” varsinaisella palautusarvolla. Jos käsittely ei onnistu, tila muuttuu hylätyksi (rejected), ja kutsutaan funktiota ”reject” poikkeuksen virhearvolla. Peräkkäisiä palvelupyyntöjä voidaan promiseilla ketjuttaa ”.then()”-metodilla, joka käsittelyn onnistumisen perusteella antaa seuraavalla pyynnölle oikean parametrin. [33][34]

Syy, miksi monadeista puhutaan paljon web-ohjelmoinnin ja varsinkin JavaScriptin näkökulmasta on se, että promise muistuttaa hyvin paljon luvussa 3.6 mainittua Maybe-monadia. Eri lähteitten mukaan promiseja pidetään suoraan monadeina, mutta ne eivät täytä kaikkia monadilakeja (luku 3.6) [2]. JavaScriptin (puoli)virallisissa dokumentaatioissa monadeja ei mainita promisejen yhteydessä, mutta yhtäläisyydet ovat suuria. Tarkastellaan niitä seuraavaksi Haskellin Maybe-monadin määritelmän kautta:

```
data Maybe a = Just a | Nothing // “|” tarkoittaa tai (or)-operaattoria
instance Monad Maybe where
(Just x) >>= k = k x
Nothing >>= k = Nothing
return = Just
```

Do-notaatiolla yksinkertainen Maybe-monadin käyttö näyttää seuraavien esimerkkien mukaiselta:

```
esimerkki1 :: Maybe Int
esimerkki1 = do
a <- Just 1
b <- Just 2
return (a + b) // palauttaa “Just 3”

esimerkki2 :: Maybe Int
esimerkki2 = do
a <- Just 1
b <- Nothing
return (a + b) // palauttaa “Nothing” [8, s. 33-34]
```

Just-arvo tarkoittaa siis onnistunutta operaatiota (fulfilled-tila promisessa) ja Nothing tarkoittaa virhettä (rejected-tila promisessa). Virhearvo Nothing etenee funktiokutsuketjussa tavallaan automaattisesti, ja näin vältytään callback-helvetti tilanteelta, jos ketjutetaan useampia funktiokutsuja.

[2] https://medium.com/javascript-scene/javascript-monads-made-simple-7856be57bfe8

JavaScriptin promisejen `.then()`-metodin käyttö muistuttaa hyvin paljon sitä, miten Haskellin `bind`-funktiolla (”`>>=`”) ketjutetaan funktiokutsuja. Haskellin `do`-notaatio auttaa ohjelmoijaa suorittamaan asioita haluamassaan järjestyksessä. Saman saa aikaiseksi `.then()`-metodilla seuraavan esimerkin tapaisesti, jossa kuvitteellisesti haetaan dataa web-sivulta, ja dataa käsitellään useammalla tavalla. Virheenkäsittely voidaan hoitaa lopussa yhdellä `catch()`-metodilla:

```
fetch(’/…/user.json’)
    .then(response => response.json())
    .then(user => fetch(`https……/${user.name]`))
    .then(response => response.json())
    .then(… => …)
    .catch(error => alert(error.message));
```

[35]

5 Yhteenveto

TK 1: Funktionaalisen ohjelmoinnin matemaattinen tausta perustuu lambdakalkyyliin ja kategoriateoriaan. Lambdakalkyyli on muodollinen laskennan malli, jolla voi ilmaista mitä tahansa matemaattisia laskuongelmia. Sitä voidaan pitää ohjelmointikielen abstraktina mallina. Sen keksi Alonzo Church vuonna 1936 ratkaisuna päätäntäongelmaan. Samana vuonna ja samaan ongelmaan, Alan Turing keksi Turingin koneen, jota voi pitää tietokoneen abstraktina mallina. Turing todisti kummankin laskentamallin olevan ekvivalentit. Turingin koneeseen perustuvat imperatiiviset ohjelmointikielet ovat paljon yleisemmässä käytössä, kuin lambdakalkyyliin perustuvat funktionaaliset ohjelmointikielet.

Lambdakalkyyli perustuu puhtaisiin funktioihin, jotka eivät aiheuta sivuvaikutuksia, kuten jotain ohjelmalle yhteisen tilan muuttamista. Sivuvaikutuksiksi lasketaan kuitenkin myös tietokoneen käytön kannalta hyvin olennaisia asioita, kuten vuorovaikutus sen käyttäjän tai verkkoliikenteen kanssa. Sivuvaikutusten hallinta puhtailla funktioilla on aiheuttanut päänvaivaa, ja se täytyy tehdä tietyllä tavalla. Nykyään käytetyimmässä puhtaasti funktionaalisessa kielessä, Haskellissa, sivuvaikutukset käsitellään monadien avulla. Monadit ovat abstraktin algebran osa-alueen kategoriateorian rakenne, joilla voidaan tehdä kuvauksia eri kategorioiden välillä niin, että sen rakenne pysyy lopulta samanlaisena. Kuvauksessa syntyvät sisäkkäiset rakenteet niin sanotusti litistetään yksinkertaisimpaan muotoonsa. Monadeille on luotu käytännön toteutus Haskell-ohjelmointikielelle.

TK 2: Funktionaalisen ohjelmoinnin, erityisesti Haskellin, erikoisominaisuuksia ovat korkea abstraktiotaso, vahva tyyppijärjestelmä, laiska suoritustapa, sivuvaikutusten hallinta monadeilla, sekä monet lopulta lambdakalkyylistä periytyvät ohjelmointitavat. Näitä ovat muun muassa currying, osittainen suoritus sekä anonyymit funktiot. Työssä esiteltiin tarkemmin, kuinka sivuvaikutusten hallinta monadeilla on viittauksellisesti läpinäkyvää: lauseke voidaan korvata sen arvolla niin, ettei ohjelman toiminta muutu. Jos funktio voidaan korvata suoraan sen palauttamalla arvolla, se ei voi esimerkiksi muuttaa jonkin globaalin muuttujan arvoa.

TK 3: Yleisemmissä kielissä käytetään varsinkin funktionaalisesta ohjelmoinnista tuttua deklaratiivista ohjelmointitapaa, jossa tietokoneen käskyttämisen sijaan sille kuvaillaan, mitä halutaan laskea. Tätä sovelletaan erityisesti iteroitaessa jotain tietorakennetta esimerkiksi funktionaalisesta ohjelmoinnista tutun map-funktion avulla. Deklaratiivinen iterointitapa vaatii laiskan suoritustavan, jota voi pitää ylimääräisenä toimenpiteenä tavallisesti ahkeralta kieleltä. Myös anonyymejä funktioita käytetään paljon yleisemmissä kielissä, ja monadien ideaa on sovellettu asynkronisten funktioiden parissa.

Funktionaalinen ohjelmointitapa on imperatiiviseen ohjelmointiin verrattuna epäsuositumpi ohjelmointitapa. Epäsuosio johtuu kenties epäintuitiivisesta tai vaikeammasta lähestymistavasta ohjelmointiin puhtaiden matemaattisten funktioiden kautta. Turingin koneeseen perustuva imperatiivinen koneen käskyttäminen vaikuttaa suosionsa perusteella olevan helpommin lähestyttävää.

Haskellin vahva staattinen tyyppijärjestelmä ja kompakti, hyvin matemaattiselta näyttävä syntaksi voivat karkottaa käyttäjiä. Oma asiansa on vielä Haskellin hyvin korkea abstraktiotaso, joka näyttäytyy varsinkin monadien parissa. Monadit itsessään abstrahoivat kolmea eri tietorakennetta. Sivuvaikutusten hallinta monadeilla vaikuttaa varmasti oudolta ja ihmeelliseltä ainakin ensi alkuun. Yksittäisten sivuvaikutusten hallinta ei välttämättä ole kovin haastavaa, mutta eri sivuvaikutusten ketjutus voi jo olla haastavampaa.

Haastavuuden vastapainona funktionaalisella ohjelmoinnilla on omat etunsa verrattuna vallitsevaan olio-ohjelmointiin. Puhtailla funktioilla tehty ohjelma voidaan perustella oikein toimivaksi, ja isojen ohjelmistojen hallinta on niiden, ja vahvan tyyppijärjestelmän, avulla paljon helpompaa. Lambdakalkyyli tukee luonnostaan rinnakkaisuutta ja Haskell tukee samanaikaisuuden hallintaa tietokannoista tutulla tekniikalla. Näillä asioilla tulee tulevaisuudessa olemaan vielä tärkeämpi rooli ohjelmistokehityksessä.

Tulevaisuuden kannalta on mielenkiintoista seurata, muuttuuko vallitseva ohjelmoinnin ajattelumalli. Tullaanko funktionaalista ohjelmointia opettamaan enemmän? Tuleeko valtavirtakieleksi jokin F#:n tapainen olio- ja funktionaalista ohjelmointia yhdistelevä kieli? Voiko Haskellista tulla valtavirtakieli? Kehitys näyttäisi hieman siihen suuntaan, että ainakin joitain funktionaalisia ominaisuuksia halutaan jo yleisesti käyttää.

Lähteet

[1] Statista.com, kyselytutkimus käytetyimmistä ohjelmointikielistä ohjelmistokehittäjien keskuudessa https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ 22.1.2024

[2] Barendregt, Henk & Barendsen, Erik, Introduction to Lambda Calculus. Revised edition 1998, March 2000

[3] Allen, Christopher & Moronuki, Julie, Haskell Programming from first principles, 2016, Lorepub LLC, ISBN-10: 194538803x

[4] Encyclopedia.com, Von Neumannin arkkitehtuurin määritelmä: https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/von-neumann-machine, vierailtu 22.1.2024

[5] MDN Web Docs, korkean tason ohjelmointikielen määritelmä: https://developer.mozilla.org/en-US/docs/Glossary/High-level_programming_language vierailtu 22.1.2024

[6] Bartosz Milewskin kotisivut, ja hänen kirjaansa Category Theory For Programmers perustuva blogi: https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/ vierailtu 22.1.2024

[7] Encyclopedia.com, sievennyskoneen määritelmä: https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/reduction-machine vierailtu 22.1.2024

[8] Diehl, Stephen, What I Wish I Knew When Learning Haskell. Blurb, Incorporated, 2020, https://smunix.github.io/dev.stephendiehl.com/hask/ tutorial.pdf (22.1.2024), ISBN: 9781714435272.

[9] Mena, Alejandro Serrano, Practical Haskell, A Real World Guide to Programming 2nd edition, 2019, Springer Science + Business Media, ISBN-13: 978-1-4842-4479-1

[10] Pierce, Benjamin C, Types and Programming Languages, 2002, MIT Press, ISBN: 0262162091

[11] Brandl, Helmut, Typed Lambda Calculus / Calculus of Constructions. Version 1.03, https://hbr.github.io/Lambda-Calculus/cc-tex/ vierailtu 22.1.2024

[12] Häsä, Jokke & Rämö, Johanna, Johdatus abstraktiin algebraan, 2015, Gaudeamus, ISBN: 9789524957564

[13] Stoica, Ovidiu Cristinel, The Standard Model Algebra May 10, 2018

[14] Barr, Michael & Wells, Charles, Category Theory for Computing Science, Prentice Hall; 1st edition (July 6, 1990); eBook (1998 Edition), ISBN-10/ASIN: 0131204866

[15] Microsoft.com, F# dokumentaatio: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/ vierailtu 22.1.2024

[16] Haskell Wiki, johdatus Haskell-kieleen: https://wiki.haskell.org/Introduction vierailtu 22.1.2024

[17] MDN Web Docs, Javascriptin määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript vierailtu 22.1.2024

[18] Python.org, Python tutoriaali: https://docs.python.org/3/tutorial/index.html vierailtu 22.1.2024

[19] MDN Web Docs, dynaamisen tyypityksen määritelmä: https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing vierailtu 22.1.2024

[20] Hutton, Graham, Programming in Haskell (second edition), Cambridge University Press, ISBN 978-1316626221

[21] Hudak, Paul; Hughes, John; Jones, Simon Peyton & Wadler, Philip. A History of Haskell: Being Lazy With Class. April 16, 2007, Proceedings of the third ACM SIGPLAN conference on History of programming languages

[22] Jones, Simon Peyton & Wadler, Philip, Imperative functional programming. October 1992, Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages

[23] Python.org, funktionaalinen ohjelmointi Pythonilla, A.M Kuchling. Release 0.32. https://docs.python.org/3/howto/functional.html vierailtu 22.1.2024

[24] MDN Web Docs, JavaScriptin määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript vierailtu 22.1.2024

[25] Github.com, Reactin julkaisuhistoria: https://github.com/facebook/react/releases?page=10 vierailtu 22.1.2024

[26] Vuejs.org, usein kysytyt kysymykset: https://vuejs.org/about/faq vierailtu 22.1.2024

[27] Python.org, iterablen määritelmä: https://docs.python.org/3/glossary.html#term-iterable vierailtu 22.1.2024

[28] MDN Web Docs, iteraatio-protokollien määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols vierailtu 22.1.2024

[29] MDN Web Docs, spread-syntaksin määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax vierailtu 22.1.2024

[30] MDN Web Docs, destrukturointi-asetuksen määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment vierailtu 22.1.2024

[31] MDN Web Docs, nuolifunktion määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions vierailtu 22.1.2024

[32] Javascript.info, callbackin määritelmä: https://javascript.info/callbacks vierailtu 22.1.2024

[33] MDN Web Docs, promisen määritelmä: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise vierailtu 22.1.2024

[34] Javascript.info, promisen määritelmä: https://javascript.info/promise-basics vierailtu 22.1.2024

[35] Javascript.info, promisejen ketjutuksen määritelmä: https://javascript.info/promise-chaining vierailtu 22.1.2024