Tampereen yliopisto

Lauri Holopainen

FUNKTIONAALISEN OHJELMOINNIN OMINAISUUDET

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Toukokuu 2019

TIIVISTELMÄ

Lauri Holopainen: Funktionaalisen ohjelmoinnin ominaisuudet
Kandidaatintyö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2019

Tässä kandidaatintyössä käsitellään funktionaalisen ohjelmoinnin ominaisuuksia. Näistä ominaisuuksista käsittelyssä on vain keskeisimmät eli puhtaat funktiot, sivuvaikutusten käsittely ja korkeamman asteen funktiot. Ominaisuudet käsitellään Scalalla tehtyjen koodiesimerkkien kautta, jotka työn kirjoittaja on tehnyt itse. Lisäksi työssä pyritään havainnollistamaan näiden ominaisuuksien ja suunnittelumallien hyötyjä ja haittoja perinteiseen oliopohjaiseen tapaan verrattuna. Tutkimuskysymyksenä on, että mitä on funktionaalinen ohjelmointi ja mitä etuja sillä on.

SISÄLLYS

1. JOHDANTO

Funktionaalisen ohjelmoinnin historia alkaa 1930-luvulla kehitetystä lambdakalkyylista. Lambdakalkyyli on formaali esitystapa matemaattisille funktioille. Siinä funktioiden mekanismi on piilotettuna ja funktioilla ei myöskään ole minkäänlaista tilaa. Funktionaaliset ohjelmointikielet noudattavat näitä periaatteita ja niiden syntaksi on monesti myös hyvin samankaltaista. Ensimmäinen funktionaalinen ohjelmointikieli oli 1950-luvulla kehitetty LISP [11]. Oliopohjaisten kielten historia alkaa vasta myöhemmin 60-luvulla, Simulan kehittämisen jälkeen [10]. Funktionaalisia ohjelmointikieliä käytetään kuitenkin opetuksessa ja ohjelmistotuotannossa huomattavasti vähemmän kuin oliopohjaisia kieliä. Tämän takia funktionaalinen ohjelmointi saattaa olla vierasta monelle kokeneellekkin ohjelmoijalle.

Tämän työn tarkoituksena on toimia yleiskatsauksena funktionaalisen ohjelmoinnin erityispiirteisiin. Tutkimuskysymyksenä on mitä on funktionaalinen ohjelmointi ja mitä etuja siinä on käytännössä.

Työssä alussa luvussa 2 käsitellään puhtaita funktioita. Luvussa 3 siirrytään sivuvaikutusten käsittelyyn puhtaiden funktioiden vaatimuksen pohjalta. Luvussa 4 käsitellään erityyppisiä korkeamman asteen funktioita ja lopussa on yhteenveto funktionaalisen ohjelmoinnin eduista ja haasteista.

2. PUHTAAT FUNKTIOT

Funktionaalisen ohjelmoinnin perusperiaate on käyttää mahdollisimman paljon puhtaita funktioita. Puhtaan funktion ainoa tehtävä on ottaa sisään parametreja ja palauttaa joku tulos. Epäpuhdas funktio saattaa tämän lisäksi tehdä myös jotain muuta. Funktio on puhdas, jos se täyttää nämä kaksi ehtoa:

1. Funktion tulos riippuu ainoastaan sen vastaanottamista parametreista.
2. Funktiolla ei ole sivuvaikutuksia.

Havainnollistetaan ensin ensimmäistä kohtaa tekemällä kaksi funktiota samalla toiminnallisuudella. Näistä ainoastaan toinen täyttää kohdan yksi ehdon. Määritetään toiminnallisuus siten että funktio ottaa parametrina kokonaisluvun ja palauttaa tuloksen, jossa tähän kokonaislukuun on lisätty viisi.

```
1   var increment : Int = 5
2   def incrementNumber(number: Int) : Int = {
3     number + increment
4   }
```

Ohjelma 1.

incrementNumber-funktio täyttää tämän toiminnallisuuden alustamalla muuttujan increment arvolla 5 ja lisäämällä sen parametrina saatuun arvoon.

```
1 def incrementNumber2(number: Int) : Int = {
2     number + 5
3 }
```

Ohjelma 2.

Molemmat funktiot tekevät täsmälleen saman asian eli palauttavat parametrinaan saaman numeron ja luvun viisi summan. Ainoastaan IncrementNumber2 funktio on puhdas. Tämä johtuu siitä, että incrementNumber-funktio riippuu parametrien lisäksi funktion ulkopuolisesta muuttujasta *increment*.

Käsitellään seuraavaksi mitä tarkoitetaan sillä, että funktioilla ei ole sivuvaikutuksia. Jos funktiolla on joku muu havaittava vaikutus kuin sen palauttama arvo, niin sillä on sivuvaikutuksia. Tällaisia vaikutuksia ovat jonkin muuttujan arvon muuttaminen ja ohjelman ulkopuolella havaittavat vaikutukset kuten tietokantaan tallennus tai tiedostoon tulostaminen. Tiivistettynä tämä tarkoittaa minkä tahansa tilan muuttamista. Jos muuttujan arvo muuttuu, sen tila muuttuu, jos tietokantaan lisätään rivi niin tietokannan taulun tila muuttuu ja jos tiedostoon lisätään rivejä, tiedoston tila muuttuu.

Tehdään tällä kertaa yksinkertainen ohjelma, jonka tehtävä on ottaa parametrina lista lukuja ja laskea niiden summan. Otetaan tällä kertaa esimerkiksi kolme funktiota, joista kukin suhtautuu eri tavalla sivuvaikutuksiin. Funktion toiminnallisuus määritellään siten, että se ottaa parametrina listan luvuista ja palauttaa listan lukujen summan.

```
1  var start : Int = 0
2  //Tällä funktiolla on sivuvaikutuksia
3  def summa1(luvut : List[Int]) : Int = {
4     var summa : Int = 0
5     while(start < luvut.length) {
6        summa = luvut(start) + summa
7        start += 1
8     }
9     System.out.println(summa)
10    //tallentaa summan tietokantaan
11    persist(summa)
12 }
```

Ohjelma 3.

Summa1-funktio sisältää useita sivuvaikutuksia. Ulkoisia sivuvaikutuksia ovat luvun tulostaminen, sen tallentaminen tietokantaan ja ulkoisen muuttujan *start* muuttaminen. Paikallisena sivuvaikutuksena on *summa* muuttujan muuttaminen. *Start*-muuttujan muuttaminen aiheuttaa sivuvaikutuksena, sen että funktio palauttaa eri arvoja riippuen, milloin sitä kutsutaan. Havainnollistetaan tätä alla olevalla ohjelmanpätkällä.

```
1  def main(args: Array[String]): Unit = {
2      val luvut : List[Int] = List(1, 2 ,3, 2, 3)
3      val luvut2 : List[Int] = List(1, 4, 2, 7, 4, 8, 5, 76, 5, 323)
4      val sum1 : Int = 11
5      val sum2 : Int = 435
6      println(sum1)
7      println(sum2)
8    }
```

Ohjelma 4.

Nyt kun tämä ohjelma ajetaan, funktiokutsu *summa1(luvut)* antaa ensin tulokseksi 11. Toisella kerralla ajettaessa tulokseksi kuitenkin tulee 0. Tämä johtuu aiemmin mainitusta sivuvaikutuksesta, jossa *start* muuttujaa kasvatetaan funktiota kutsuessa. Toisella kutsukerralla se on kasvanut suuremmaksi kuin luvut-listan pituus ja tällöin while-lohkon sisällä olevaa koodia ei kutsuta kertaakaan.

Tässä on havainnollistettuna yksi syy, miksi sivuvaikutuksia pyritään välttämään. Sivuvaikutusten ongelma on siinä, että ne vaikuttavat ohjelman tilaan. Tämä on suoraa seurausta sivuvaikutusten määritelmästä. Jos funktiokutsun tulos riippuu ohjelman tilasta, sitä on vaikeaa testata. Nyt jos kirjoittaisimme yksikkötestin, testaamaan palauttaako summa1-funktio oikean tuloksen listalle *luvut,* niin testi menisi läpi, koska funktio palauttaisi ensimmäisellä kutsukerralla oikean luvun 11.

```
1  // Tällä funktiolla on paikallisia sivuvaikutuksia
2  def summa2(luvut : List[Int]) : Int = {
3     var summa : Int = 0
4     for(luku <- luvut ) {
5        summa = summa + luku
6     }
7     summa
8  }
```

Ohjelma 5.

Funktio summa2 sisältää pelkästään paikallisia sivuvaikutuksia. Sen suorituksen aikana muutetaan muuttujan *summa* arvoa kun lukulistaa käydään läpi. Lisäksi *luku*-muuttuja muuttuu jokaisella for-silmukan kierroksella. Kuitenkaan sivuvaikutukset eivät näy mitenkään ulospäin. Tämä johtuu siitä, että *summa*- muuttujaan päästään käsiksi ainoastaan funktion sisältä. Silloin kun funktion sisäinen tila muuttuu, puhutaan paikallisista sivuvaikutuksista.

```
1  //Tällä funktiolla ei ole sivuvaikutuksia
2  def summa3(luvut : List[Int], summa : Int = 0, index : Int = 0) :
3  Int = {
4     if(index == luvut.length) {
5        summa
6     } else {
7        summa3(luvut, summa + luvut(index), index +1)
8     }
9  }
```

Ohjelma 6.

Funktiolla summa3 ei ole lainkaan sivuvaikutuksia. Se on rekursiivinen funktio, jota kutsuessa mikään funktion sisäinen tai ulkopuolinen tila ei muutu. Toiminnallisuus on identtinen funktio neljään nähden. Funktiossa viisi on vain kierretty sivuvaikutusten käyttö,

siten että ei käytetä muuttujia vaan kutsutaan funktiota aina kasvavalla arvolla. Onko funktio viisi siis funktionaalisempi ja parempi tapa toteuttaa lukujen summaaminen?

Molemmissa funktioissa on kuusi riviä, mutta funktio summa2 on silti huomattavasti helpolukuisempi. Se ottaa vastaan vähemmän parametreja ja siinä ei tarvitse asettaa oletusparametreja. Myöskin for-loopin toiminta on yleisesti tunnettu, kun taas funktio viidessä voi kulua hieman aikaa hahmottaa mitä siinä tapahtuu.

Funktiolla summa3 ei ole myöskään minkäänlaisia etuja funktio summa2:een nähden. Paikalliset sivuvaikutukset ovat hyväksyttäviä funktionaalisessa ohjelmoinnissa [1, s.256]. Sivuvaikutusten eliminoimisen ideana on se, että ne eivät aiheuttaisi odottamattomia vaikutuksia muun ohjelman toimintaan. Tästä nähtiin yksinkertainen esimerkki funktio kolmen tapauksessa.

Lisäksi lausekkeet ovat molemmat viitteellisesti läpinäkyviä (eng. referential transparency). Funktio on viitteellisesti läpinäkyvä, jos funktiokutsu voidaan korvata sen palauttamalla arvolla, ilman että ohjelman toiminta muuttuu millään tavalla. Summa3- ja Summa2-funktiot täyttävät tämän ehdon [1, s. 10-12]. Havainnollistetaan tätä vielä lyhyellä esimerkillä.

```
1  def main(args: Array[String]): Unit = {
2     val luvut : List[Int] = List(1, 2 ,3, 2, 3)
3     val luvut2 : List[Int] = List(1, 4, 2, 7, 4, 8, 5, 76, 5, 323)
4     val sum1 : Int = summa2(luvut)
5     val sum2 : Int = summa3(luvut2)
6     println(sum1)
7     println(sum2)
8  }
```

Ohjelma 7.

Ohjelmassa 7 on lisätty sivuvaikutuksiksi laskettujen summien tulostus. Se tulostaa luvut, 11 ja 435. Tehdään nyt toinen ohjelma, jossa sum1 ja sum2 funktiokutsut korvataan niiden palauttamalla arvolla.

```
1  def main(args: Array[String]): Unit = {
2     val luvut : List[Int] = List(1, 2 ,3, 2, 3)
3     val luvut2 : List[Int] = List(1, 4, 2, 7, 4, 8, 5, 76, 5, 323)
4     val sum1 : Int = 11
5     val sum2 : Int = 435
6     println(sum1)
7     println(sum2)
8  }
```

Ohjelma 8.

Huomataan, että koodi 7 ja koodi 8 tekevät täysin saman asian, eli tulostavat näytölle luvut 11 ja 435. Jo aikaisemmin oli todettuna, että summa2- ja summa3-funktioilla ei ole ulkoisia sivuvaikutuksia. Näin ollen ne ovat myös viitteellisesti läpinäkyviä ja funktiokutsut pystytään korvaamaan niiden palauttamilla arvoilla.

3. SIVUVAIKUTUSTEN KÄSITTELY

Edellisessä luvussa totesimme, että puhtaat funktiot eivät sisällä sivuvaikutuksia. Kuitenkin jos halutaan tehdä minkäänlaista käyttökelpoista ohjelmistoa, tarvitaan sivuvaikutuksia. Ilman mitään sivuvaikutuksia emme saa tallennettua dataa tai näytettyä sitä mitenkään käyttäjälle. Funktionaalisessa ohjelmoinnissa ei ole tavoitteena vähentää sivuvaikutusten määrää keinotekoisesti. Tässä luvussa käymme läpi millä tavalla funktionaalisissa ohjelmistoissa on eriytetty laskennallinen logiikka ja sivuvaikutukset toisistaan.

Funktionaalisen ohjelman tulisi olla kauttaaltaan viitteellisesti läpinäkyvä. Jokainen ohjelman funktiokutsu tulisi olla korvattavissa sen palauttamalla arvolla. Tämä tapahtuu käytännössä, siten että ohjelman main-metodi palauttaa suoritettavan ohjelman arvona.

3.1. Proseduurit

Funktionaalisessa ohjelmoinnissa proseduurit ovat funktioita tai metodeja, joilla on sivuvaikutuksia. Proseduurit palauttavat yleensä vain tyhjän arvon, joka on Scalassa tyyppiä Unit. Koska palautettu arvo ei sisällä mitään merkitystä, proseduurin ainoa toiminnallisuus on sen tekemissä ulkoisissa sivuvaikutuksissa ja ohjelman tilan muuttamisessa. Yksi tapa eriyttää ohjelman laskennallinen logiikka ja sivuvaikutukset toisistaan on siis jakaa ohjelma puhtaisiin funktioihin ja proseduureihin [1, s. 17].

Aiemmin koodissa 3 loimme funktion summa1, joka sekä laski lukujen summan, tulosti sen ja tallensi sen tietokantaan. Totesimme että tämä ei ole puhdas funktio, joten sellaista ei tulisi tehdä funktionaaliseen ohjelmaan. Meillä voi kuitenkin olla tarve summan laskemisen lisäksi myös tallentaa luku tietokantaan ja tulostaa se käyttäjälle näkyviin. Tehdään nyt funktionaalinen esimerkki, jossa summa1:n toiminnallisuus jaetaan proseduureihin ja puhtaisiin funktioihin.

```
  def summa2(luvut : List[Int]) : Int = {
    var summa : Int = 0
    for(luku <- luvut ) {
      summa = summa + luku
    }
    summa
  }

  def saveToDB(summa: Int) : Unit = {
    System.out.println(summa)
    persist(summa)
  }

  def main(args: Array[String]): Unit = {
    val luvut : List[Int] = List(1, 2, 3, 2, 3)
    saveToDB(summa2(luvut))
}
```

Ohjelma 9.

Nyt ohjelma koostuu main-funktiosta, proseduurista sekä puhtaasta funktiosta. Summan laskeva funktio on puhdas funktio, jolla on vain paikallisia sivuvaikutuksia. Huomataan myös, että koko ohjelma on nyt viitteellisesti läpinäkyvä. Itse ohjelmassa ei myöskään ole sivuvaikutuksia, vaikka proseduuri saveToDB niitä sisältääkin. Main-funktio palauttaa suoritettavan ohjelman arvona, joka sitten suoritetaan sivuvaikutuksineen. Itse ohjelma ei sisällä sivuvaikutuksia vaan ainoastaan kuvauksen niistä.

Yksi tapa erottaa selkeästi ja yksiselitteisesti laskennallinen logiikka ja sivuvaikutukset toisistaan on siis jakaa ohjelma proseduureihin ja puhtaisiin funktioihin. Proseduurit palauttavat aina Unit- tyypin ja puhtaat funktiot aina jonkun arvon.

3.2. Monadit

Yksi tapa käsitellä sivuvaikutuksia on monadit. Monadi on suunnittelumalli, jossa sivuvaikutusten aiheuttama toiminto kuvataan joksikin tietotyypiksi. Termi on peräisin mate-

matiikan kategoriateoriasta. Siinä se tarkoittaa funktoria joka kuvaa kategorian itseensä. Monadi on toteutus jollekin minimaaliselle joukolle tiettyjä operaatioita. Nämä operaatiojoukot ovat

1. unit ja flatMap
2. unit ja compose
3. unit, map ja join .

[1, s. 199]

Tämän lisäksi monadin on täytettävä assosiatiivisuus ja identiteettisäännöt. Operaatio flatMap täyttää assosiatiivisuus säännön, jos seuraava yhtälö pitää paikkansa.

```
1  x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

Ohjelma 10. *Koodiesimerkki lähteestä 1, sivu 196*

Tämän täytyy päteä kaikille x, f ja g arvoille. Tämä tarkoittaa sitä, että operaatio antaa saman tuloksen riippumatta siitä, miten se on ryhmitelty. Monadiset operaatiot noudattavat tätä sääntöä samalla tavoin kuin kerto- ja yhteenlaskussa [5].

(74 + 50) + 80 = 204
74 + (50 + 80) = 204
(5 * 6) * 4 = 120
5 * (6 * 4) = 120

Jokaisella monadilla täytyy olla joku identiteettialkio. Kun identiteettialkio ja johonkin toiseen alkioon tehdään monadinen operaatio, palautuu aina tämä toinen alkio. [6] FlatMap-operaatioon tapauksessa se tarkoittaa, että seuraavat lausekkeet ovat tosia.

```
1  flatMap(x)(unit) == x
2  flatMap(unit(y))(f) == f(y)
```

Ohjelma 11. *[1, s. 197]*

Sana unit viittaa tässä kontekstissa identiteettiin. Funktionaalisessa ohjelmoinnissa sivuvaikutukset voidaan esittää palauttamalla Unit-tyypin sijaan IO-monadi. IO-monadin palauttavia funktioita voidaan ketjuttaa monadisilla operaatioilla ja näin kuvata sivuvaikutuksia sisältävä toiminnallisuus funktionaalisesti. [1, s. 231-236]

4. KORKEAMMAN ASTEEN FUNKTIOT

Funktionaalisessa ohjelmoinnissa funktioita voidaan käyttää samalla tavalla kuin muitakin datatyyppejä. Funktiot voivat ottaa siis parametrinaan funktioita ja myös palauttaa niitä. Jos funktio ottaa parametrina funktion tai palauttaa sellaisen, kutsutaan sitä korkeamman asteen funktioksi. Tämä on yksi funktionaalisen ohjelmoinnin hyödyllisimmistä omaisuuksista [4, s.34]

Korkeamman asteen funktiot ovat monikäyttöisempiä ja geneerisempiä kuin normaalit funktiot. Funktion ollessa parametrina saavutetaan erilaisia hyötyjä verrattuna tilanteeseen, jossa funktiota käytetään paluuarvona.

4.1. Funktio parametrina

Kun korkeamman asteen funktio ottaa parametrina toisen funktion, voidaan sen toiminnallisuutta muokata hyvin vapaasti. Scalassa ainoana rajoitteena on parametrina otettavalle funktiolle se, että sen parametrit ja paluuarvot määrätään korkeamman asteen funktiossa. Tämän rajoitteen puitteissa voidaan korkeamman asteen funktion toiminnallisuutta vaihtaa vapaasti. [1, s.22]

Käytetään esimerkkinä jonkun kaupan hallintajärjestelmää. Siinä tarvitaan ominaisuutta, jolla muokataan jonkun tuotteen hintaa. Tuotteen hintaa voidaan haluta lisätä jollakin vakioluvulla, sitä voidaan haluta kasvattaa jollakin prosentilla tai sen hinta voidaan haluta yhtenäistää jonkun toisen tuotteen kanssa.

```
1  def changePrice(product : String, f: Int => Int): Unit = {
2     val currentPrice : Int = getCurrentPrice(product)
3     //tallentaa tuotteelle uuden hinnan tietokantaan
4     persist(product, f(currentPrice))
5  }
```

Ohjelma 12. *proseduuri joka muokkaa tuotteen hintaa*

Ohjelma 10:n funktio *changePrice* ottaa parametrinaan toisen funktion, joka määrää millä logiikalla tuotteen hintaa muutetaan. Tuotteen hintaa voidaan haluta alentaa alennusmyyntien takia tietyllä prosentilla. Korkeamman asteen funktio *changePrice,* ottaa silloin parametrinaan funktion, joka laskee alennetun hinnan.

```
1    def discount(number : Double ) : Double = {
2      //getCurrentDiscount hakee alennuskertoimen tietokannasta
3      number * getCurrentDiscount()
4    }
5
6    changePrice("tuote1", discount )
```

Ohjelma 13. *Korkeamman asteen funktion kutsuminen*

Jos tiettyä logiikkaa hinnan muokkaukseen käytetään vain yhdessä paikassa koko ohjelmassa, voidaan korkeamman asteen funktiolle antaa parametriksi myös anonyymi funktio.

```
1  changePrice("Tuote2", (x: Double) => x+5.0)
```

Korkean asteen funktiota on siis yleisesti mahdollista käyttää useammalla eri tavalla. Tämä lisää ohjelman modulaarisuutta ja myös vähentää tarvittavien koodirivien määrää. Korkean asteen funktioita ei myöskään yleensä tarvitse muokata ohjelman vaatimusten kehittyessä, sillä lisävaatimukset voidaan täyttää käyttämällä parametrina eri funktioita.

4.2. Funktio paluuarvona

Laajempaa ohjelmaa tehtäessä voi tulla tilanne, jossa tietyssä kohtaa ohjelman suoritusta seuraava toiminnallisuus riippuu ohjelman tilasta. Oliopohjaisella lähestymistavalla tämä ratkaistaisiin useammalla if-lohkolla tai käyttämällä case-rakennetta. Nämä rakenteet ovat sallittuja myös funktionaalisessa ohjelmoinnissa, mutta ongelman voi ratkaista myös korkeamman asteen funktiolla. Jos seuraava suoritettava logiikka riippuu ohjelman tilasta, voidaan tehdä korkeamman asteen funktio, joka palauttaa tarvittavan toiminnallisuuden toteuttavan funktion.

Laajennetaan aiemmin esitettyä kaupanhallintajärjestelmää. Se miten hintaa muutetaan riippuu ohjelman tilasta, esimerkiksi alennukset saattavat olla viikonloppuisin ja arkisin erilaiset.

```
1    def generatePriceChangeFunction(): Double => Double = {
2      val factor : Double = if(weekend)  0.75 else 1.0
3      (number : Double) => factor * discount(number)
4    }
5
6  raisePrice("Tuote2", generatePriceChangeFunction())
```

Ohjelma 14.

Nyt alennuksen määräävä funktio on erilainen riippuen viikonpäivästä. Kaupassa voi olla käytössä päällekkäin monenlaisia eri alennuksia. Korkeamman asteen funktiolla saavutimme sen, että näitä voi laskea päällekkäin ongelmitta.

5. YHTEENVETO

Funktionaalisessa ohjelmoinnissa on paljon hyödyllisiä piirteitä ja suunnittelumalleja, jotka tarjoavat konkreettisia hyötyjä ohjelmien kehitykseen. Se mahdollistaa tietyissä tapauksissa modulaarisemman ja helpommin testattavissa olevan ohjelman kehittämisen verrattuna olio-ohjelmointiin. Täysin ongelmaton ohjelmointiparadigma se ei kuitenkaan ole.

Vaatimukset puhtaista funktioista ja ohjelman sivuvaikutusten kuvaamisessa tietyllä tavalla aiheuttavat haasteita ohjelmoijalle. Sivuvaikutuksia kuitenkin tarvitaan ohjelmassa, mutta niiden toteuttamiseen funktionaalisella ohjelmoinnilla on haastavaa, johtuen ylimääräisistä vaatimuksista. Myöskin se, että muuttujan arvoja ei tulisi muuttaa ohjelman aikana voi tuntua aluksi erittäin epäintuitiiviselta. Oliopohjaisissa ohjelmissa yksi olio kuvaa usein jotain reaalimaailman käsitettä. Esimerkiksi yksi käyttäjä voidaan kuvata yhdeksi olioksi. Oikean maailman käyttötapauksissa käyttäjän tila muuttuu ajan myötä. Funktionaalisessa ohjelmoinnissa joudutaan muutoksen jälkeen luomaan uusi olio kuvaamaan muuttunutta tilannetta. Oliopohjaisessa lähestymistavassa sen sijaan olion tilaa muutetaan. Olion konsepti on siis näissä paradigmoissa hyvin erilainen, mikä voi aiheuttaa hämmennystä ohjelmoijalle.

Monadien konseptin ymmärtämiseen ja sen käyttöön ohjelmoinnissa on tehty paljon ohjeita [7][8]. Tämä kertoo siitä, että se on tärkeä konsepti funktionaalisessa ohjelmoinnissa. Se kertoo myös todennäköisesti siitä, että sen hahmottaminen on monille vaikeaa.

Lähteestä riippuen funktionaalisella ohjelmoinnilla tarkoitetaan eri asioita eri painotuksilla, mikä myöskin kasvattaa oppimiskynnystä. Osassa lähteistä puhtaissa funktioissa ei sallita paikallisia sivuvaikutuksia vaan ne pitäisi korvata rekursiolla. Osassa taas nähdään, että paikalliset sivuvaikutukset ovat sallittuja, sillä niillä ei ole vaikutusta ohjelman tilaan. Tässä varmasti vaikuttaa se, että tietyissä funktionaalisissa ohjelmointikielissä paikalliset sivuvaikutukset eivät ole edes teknisesti mahdollisia. Yksi esimerkki on Haskell [9].

LÄHTEET

[1] Paul Chiusano and Runar Bjarnason - Functional Programming in Scala, 2014, Saatavissa: https://www.manning.com/books/functional-programming-in-scala

[2] Scalan kotisivut ja dokumentaatio. Saatavissa: https://www.scala-lang.org/ ,

[3] https://www.computerweekly.com/feature/Write-once-run-anywhere

[4] http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf , Purely Functional Data Structures, Chris Okasaki, 1996

[5] https://www.computerhope.com/jargon/a/assooper.htm

[6] Identity element, https://brilliant.org/wiki/identity-element/

[7] https://wiki.haskell.org/Monad_tutorials_timeline , Monad tutorials timeline, HaskellWiki, Marraskuu 2015'

[8] https://alvinalexander.com/programming/functional-programming-collection-good-monad-tutorials, Functional programming collection good monad tutorials, Tammikuu 2018

[9] Why Haskell matters, Lokakuu 2015, https://wiki.haskell.org/Why_Haskell_matters

[10] J.Sklenar, Introduction to OOP in Simula, 1997http://staff.um.edu.mt/jskl1/talk.html

[11] John McCarthy, History of Lisp, Helmikuu 1979 http://jmc.stanford.edu/articles/lisp/lisp.pdf