Tampereen yliopisto

Ville Werner

MONADIEN KÄYTTÖ HASKELL-OHJELMOINTIKIELESSÄ

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Petri Kannisto
Huhtikuu 2021

TIIVISTELMÄ

Ville Werner: Monadien käyttö Haskell-ohjelmointikielessä
Kandidaatintyö
Tampereen yliopisto
Tutkinto-ohjelma
Huhtikuu 2021

Tässä työssä tutkittiin monadeja sekä sitä, mitä hyötyä niistä on ohjelmoijalle. Monadeja tarkasteltiin lähtökohtaisesti Haskell-ohjelmointikielen kannalta.

Ensiksi käytiin läpi, mitä monadi tarkoittaa kategoriateoriassa. Havaintona oli, että monadi on funktorista ja kahdesta luonnollisesta transformaatiosta muodostuva kolmikko, joka täyttää tietyt sille asetetut ehdot.

Seuraavaksi selvitettiin, miten monadin määritteleminen onnistuu Haskellissa. Tämä oli suhteellisen suoraviivaista. Funktori korvattiin sekä operaattorilla, joka tuottaa sille annetusta arvosta monadisen, että funktiolla $fmap$, jolla alkuperäistä arvoa voidaan muuttaa. Lisäksi luonnolliset transformaatiot korvattiin funktioilla, joista toinen tuottaa triviaalin kontekstin minkä tahansa arvon ympärille ja toinen yhdistää kaksi kontekstia yhdeksi.

Tämän jälkeen esiteltiin viisi erilaista yleisesti käytettyä monadia. Näistä monadeista käsiteltiin esimerkiksi sitä, miten niiden avulla saa liitettyä yhteen funktioita siten, että triviaali toiminnallisuus toteutetaan implisiittisesti. Tällöin kirjoitettu koodi on tiiviimpää ja keskittyy enemmän siihen, mikä on kyseisten funktioiden toiminnassa keskeistä.

Lopuksi tarkasteltiin kahta eri tapaa, joilla monadien tarjoamaa toiminnallisuutta saadaan laajennettua.

Avainsanat: monadi, funktionaalinen ohjelmointi

SISÄLLYSLUETTELO

1. JOHDANTO

Funktionaalisessa ohjelmoinnissa ongelmia käsitellään eri lähtökohdista kuin muissa ohjelmointiparadigmoissa. On luontevaa, että myös koodin jäsentelyyn käytetyt menetelmät ovat omanlaisiaan. Yksi näistä menetelmistä on monadien käyttäminen. Tässä työssä selvitetään, mitä monadit ovat ja mitä hyötyä niistä on erityisesti Haskell-ohjelmointikielessä.

Yksi motivaatio, joka helpottaa monadin käsitteen ymmärtämistä sekä syitä sen taustalla liittyy puhtaiden ja epäpuhtaiden funktioiden väliseen eroon. Puhdas funktio toimii aina samalla tavalla, jos sille annetut parametrit ovat samat. Se ei voi vaikuttaa muun ohjelman toimintaan mitenkään muuten kuin palauttamansa arvon kautta. Epäpuhtaiden funktioiden toteutus taas on riippuvainen jostain funktion rungon ulkopuolisesta tiedosta tai funktio muokkaa tällaista tietoa muutoin kuin paluuarvonsa kautta. Epäpuhtaat funktiot voivat esimerkiksi toimia eri tavalla erilaisissa ohjelmointiympäristöissä tai niitä voidaan käyttää muokkaamaan tai poistamaan jonkin olion attribuutteja. [1]

Yleisesti ottaen epäpuhtaiden funktioiden toiminnallisuus saadaan suoritettua myös puhtailla funktioilla, jos funktion ulkopuolinen tieto sisällytetään kyseisen funktion parametreihin sekä mahdollisesti myös sen palauttamaan arvoon. Näitä muokattuja funktioita pystytään nyt yhdistelemään paljolti samankaltaisesti kuin niiden yksinkertaisempia vastineitakin. Tämä kuitenkin monimutkaistaa koodia, vaikka itse yhdistäminen olisikin monissa tilanteissa triviaalia. Monadi tarjoaa rajapinnan, jota voi käyttää muokattujen funktioiden yhdistämiseen kunhan yhdistettävät funktiot ovat samalla tavalla muokattuja. [2]

Koska monadien käsite on alunperin lähtöisin matematiikasta, tarkemmin kategoriateorian osa-alueelta, ensiksi tämä työ käsittelee monadien matemaattista luonnetta. Tästä siirrytään siihen, miten monadinen rakenne saadaan toteutettua Haskellissa. Seuraavaksi käydään läpi käytännön esimerkkejä erilaisista yleisesti käytetyistä monadeista. Lopuksi käsitellään sitä, miten monadeista saadaan rakennettua suurempia kokonaisuuksia.

2. MONADIT KATEGORIATEORIASSA

Kirjallisuudessa [3, 4] mainitaan, että monadit ovat monoideja endofunktoreiden kategoriassa. Tässä luvussa selvitetään, mitä tämä väite tarkoittaa.

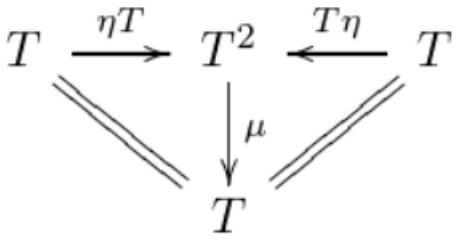
Kategoriateoria tutkii matemaattisia objekteja sekä niiden välisiä suhteita. Jokaisella kategorialla on operaatio, jolla kuvaa objektin toiseksi. Tätä operaatiota kutsutaan morfismiksi. Lisäksi jokaisella kategorian objektilla on identiteettimorfismi, joka kuvaa objektin siksi itsekseen. Morfismeja voidaan yhdistää toisiinsa. Jos morfismi f kuvaa objektin X objektiksi Y ja morfismi g puolestaan kuvaa objektin Y objektiksi Z, yhdistämällä ne saadaan morfismi gf, joka kuvaa objektin X objektiksi Z. [3]

Funktorin voi kuvitella olevan morfismin yleistys kategorioille. Koska kategoria koostuu sekä objekteista että morfismeista, funktorin pitää osata kuvata niitä molempia kahden eri kategorian välillä. Funktorit ylläpitävät kuvaamansa kategorian rakenteen [3]. Endofunktori on erityistapaus funktoreista. Sen lähtö- ja maalikategoriana toimii sama kategoria.

Monoidi taas on kokoelma objekteja, joille on olemassa binäärioperaattori, joka ottaa kaksi objektia ja palauttaa kolmannen. Myös palautettu objekti kuuluu alkuperäiseen kokoelmaan. Tämän binäärioperaattorin on oltava assosiatiivinen. Lisäksi tällä operaatiolla on oltava neutraalialkio. Jos annettua binäärioperaatiota käytetään siihen ja mihin tahansa annetun kokoelman objektiin a, saadaan tulokseksi a. [4]

Kategoriassa X oleva monadi on kolmikko (T, η, μ), missä T on endofunktori, η on luonnollinen transformaatio identiteettifunktorilta T:lle ja μ on luonnollinen transformaatio, joka kuvaa funktorin T^2 funktoriksi T. [3] Koska tällainen kolmikko muodostaa monoidin, saadaan kaksi ehtoa, jotka sen tulee täyttää.

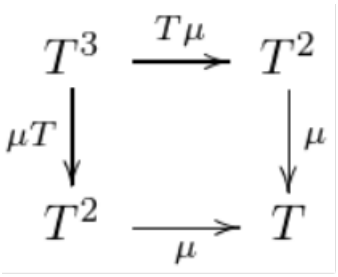
Ensimmäinen ehto liittää yhteen kaksi eri tapaa kuvata funktori T funktorin T^2 kautta takaisin itsekseen. Tämä ehto määrää, että η on neutraalialkio transformaation μ kannalta. Funktori T saadaan kuvattua funktoriksi T^2 käyttämällä transformaatiota ηT, ja saatu funktori saadaan edelleen kuvattua takaisin funktoriksi T käyttämällä transformaatiota μ. Toisaalta funktori T saadaan kuvattua funktoriksi T^2 myös transformaatiolla $T\eta$. Jälleen saatu funktori saadaan kuvattua takaisin funktoriksi T transformaatiolla μ. Ensimmäinen ehto sanoo, että nämä kaksi tapaa, joilla funktori T kuvataan funktoriksi T^2, ovat identtiset. Tämä ehto voidaan kuvata myös alla esitetyn kuvan avulla.



Kuva 2.1. *η neutraalialkiona*

[3]

Toinen ehto liittää yhteen kaksi eri tapaa kuvata funktori T^3 funktorin T^2 kautta funktoriksi T. Sen lähtökohtana on, että funktoria T^3 eli funktorin T kolminkertaista käyttämistä peräkkäin voi ajatella kahdella tavalla. Ensiksi voidaan käyttää funktoria T kaksi kertaa peräkkäin ja yhdistää se sitten vielä kerran itseensä, $T \circ T^2$. Toisaalta voidaan myös ensiksi käyttää funktoria T vain kerran ja sen jälkeen yhdistää se funktoriin T^2, $T^2 \circ T$. Jos näistä kahdesta eri lähtökohdasta päädytään samaan lopputulokseen, funktorin T täytyy olla assosiatiivinen. Tämä ehto voidaan kuvata myös alla esitetyn kuvan avulla.



Kuva 2.2. *Funktorin T itseensä yhdistämisen assosiatiivisuus*

[3]

Funktori T^3 saadaan kuvattua funktoriksi T^2 käyttämällä transformaatiota μT. Tässä on käytetty merkintää, jossa T tarkoittaa luonnollista transformaatiota funktorilta T sille itselleen. Eli alkuperäistä funktoria T^3 on käsitelty muodossa $T^2 \circ T$. Tästä saadaan edelleen muodostettua funktori T käyttämällä transformaatiota μ. Toisaalta funktori T^3 saadaan kuvattua funktoriksi myös transformaatiolla $T\mu$. Tässä tapauksessa alkuperäistä funktoria on käsitelty muodossa $T \circ T^2$. Saatu funktori saadaan taas kuvattua funktoriksi T käyttämällä transformaatiota μ. Toinen ehto sanoo, että edellä mainitut tavat kuvata funktori T^3 funktoriksi T ovat identtiset.

3. TEORIASTA OHJELMOINTIIN

Jotta ohjelmointikielessä saadaan luotua monadi, täytyy luvussa 2 kuvatut kolme operaatiota toteuttaa jollakin tavalla. Haskell-ohjelmointikielessä monadit on toteutettu omana luokkanaan [5], minkä takia se valikoitui työssä käytettäväksi kieleksi.

Edellä mainitut kolme operaatiota ovat suoraviivaisia toteuttaa. Funktorista T tulee operaattori M, joka tuottaa annetusta tyypistä sitä vastaavan monadin. Samalla toteutetaan myös funktio map, jonka avulla monadin sisältämiä arvoja voidaan muokata. Kuvauksesta η tulee funktio $unit$ tai toiselta nimeltään $return$. Tämä funktio korottaa annetun muuttujan monadiksi. Kuvauksesta μ tulee funktio $join$. Sen tehtävänä on liittää yhteen kaksi perättäistä monadikerrosta. [5]

Tämä monadin toteutus ei kuitenkaan ole se, jota Haskell-ohjelmointikielessä käytetään. Siinä käytetään funktiota $>>=$ tai toiselta nimeltään $bind$, joka voidaan määritellä funktorin T ja kuvauksen η toteutusten avulla [6]. Näin funktion $>>=$ toiminnaksi saadaan kaksivaiheinen operaatio. Ensin olemassa olevaa monadista arvoa muokataan funktiolla, joka tuottaa uuden monadisen arvon. Tässä vaiheessa uusi arvo on kahden monadikerroksen sisällä. Lopuksi perättäiset monadikerrokset yhdistetään, jolloin jäljelle jää uusi monadinen arvo. Näin saadaan monadille toinen, yhtäpitävä määritelmä, jota myös Haskell käyttää [6]. Sen oleellisimmat funktiot on mainittu alla.

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

[7]

Intuition kannalta monadista arvoa ma voi ajatella arvona (a) jossakin kontekstissa (m). Mikäli ohjelmoijalla on käytössään funktio $f :: a- > mb$, joka tuottaa yksittäisestä annetusta arvosta (a) uuden monadisen arvon (mb). Nyt funktion $>>=$ tehtävänä on tuottaa uusi monadinen arvo alkuperäisen monadisen arvon sisältämästä arvosta. Tämän lisäksi se yhdistää alkuperäisen ja funktion f avulla saadun kontekstin yhdeksi uudeksi kontekstiksi. Eri monadien kontekstit eroavat yleensä paljolti toisistaan. Siksi myös funktion $>>=$ toteutukset eroavat toisistaan. [2]

4. KÄYTÄNNÖN ESIMERKKEJÄ MONADEISTA

Tämä luku esittelee muutamia yleisesti käytettyjä monadeita. Esiteltävistä monadeista huomataan esimerkiksi, miten niiden avulla saadaan yhdistettyä puhtaita funktioita, jotka toteuttavat alun perin epäpuhtaan toiminnallisuuden. Lisäksi huomataan, miten monadien käyttäminen selkeyttää kirjoitettua koodia.

4.1 Maybe

Alla olevassa koodinpätkässä 4.1 esitellään sekä tietotyypin $Maybe$ että sen monadisen rajapinnan toteutus.

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    (Just x) >>= k   =   k x
    Nothing  >>= k   =   Nothing
    return           =   Just
```

Kuva 4.1. *Maybe-tietotyypin ja sen monadisen rajapinnan toteutus* [7]

Maybe-monadia käytetään, kun halutaan tarkistaa, onnistuiko jonkin operaation suoritus, ennen uusien operaatioiden suorittamista. Tämän voi tehdä ehtolausekkeilla, mutta etenkin ketjutettaessa useita operaatioita peräkkäin kaikkien tarvittavien ehtolausekkeiden kirjoittaminen tekee koodista vaikeammin tulkittavaa. Käytettäessä Maybe-monadia operaatiot saadaan ketjutettua käyttämällä funktiota $>>=$. Tällöin onnistuneen operaation tapauksessa saatua arvoa käytetään edelleen seuraavien operaatioiden lähtöarvona. Epäonnistuneen operaation jälkeen taas kaikki jäljellä olevat operaatiot hylätään ja koko ketjun tulokseksi saadaan epäonnistumisesta kertova arvo $Nothing$. [8]

Otetaan esimerkiksi tietokanta, joka sisältää tietoa työntekijöiden esimiehistä. Oletetaan, että kullakin työntekijällä on enintään yksi lähin esimies. Koodinpätkässä 4.2 on esitetty yksi esimerkki annetun työntekijän lähimmän esimiehen ja esimiehen esimiehen etsimisestä C++ -ohjelmointikielellä.

Molemmat esitetyistä funktioista ovat selvästi epäpuhtaita. Niiden toiminta riippuu glo-

```
using std::map
using std::string

map<string,string> tietokanta = ...;

string lahin_esimies(string tyontekija)
{
    return tietokanta[tyontekija];
}

string esimiehen_esimies(string tyontekija)
{
    string lesimies = tietokanta[tyontekija];
    return tietokanta[lesimies];
}
```

Kuva 4.2. *Esimiesten etsiminen C++ -ohjelmointikielellä*

baalisti määritellystä tietorakenteesta $tietokanta$. Riippuen siitä, löytyykö haettu työntekijä funktion kutsumishetkellä tietorakenteesta, tietorakenteen indeksointi joko palauttaa kyseisen työntekijän esimiehen tai vuotaa poikkeuksen.

Koodinpätkässä 4.3 on toteutettu puhtaat versiot funktioista $lahin\_esimies$ ja $esimiehen\_esimies$. Ne käyttävät apunaan funktiota $etsi$. Mikäli annettua avainta vastaa jokin arvo annetussa tietokannassa, $etsi$ palauttaa avainta vastaavan arvon. Muussa tapauksessa ohjelma kaatuu.

Jotta saadaan toteutettua turvallinen funktio $lahin\_esimies$, jolla ohjelman kaatuminen saadaan estettyä, tutkitaan ensin, löytyykö annettu avain tietokannasta. Vasta sen jälkeen käytetään funktiota $etsi$ tai vaihtoehtoisesti kerrotaan, että annettua avainta ei löytynyt.

Jos lähimmän esimiehen sijasta halutaankin saada selville annetun työntekijän esimiehen esimies, toimitaan täsmälleen samoin, kuin funktiossa $lahin\_esimies$, mutta tällä kertaa edellä esitettyä logiikkaa käytetään kahdesti. Ensin tutkitaan, löytyykö lähin esimies tietokannasta, ja sen jälkeen, löytyykö myös esimiehen esimies. Mikäli jompaakumpaa näistä alaisen ja esimiehen välisistä suhteista ei löydy, kerrotaan, että etsittyä tietoa ei löytynyt.

Toistamalla tätä toimintatapaa saadaan etsittyä mielivaltaisen korkealla esimieshierarkiassa olevia henkilöitä. Kuitenkin jo esimiehen esimiestä etsittäessä koodista ei välttämättä saa ensinäkemältä selvää. Tilanne vain pahenee, mitä ylempää esimieshierarkiasta henkilöitä etsitään. Kuitenkin epäonnistumisesta kertovan arvon palauttaminen tilanteissa, joissa haluttua operaatiota ei saataisi suoritettua, on triviaalia.

Esimiesten etsintä saadaan muokattua helpommin luettavaan muotoon käyttämällä maybemonadia. Funktio $lookup$ on toiminnaltaan samanlainen kuin funktio $etsi$, mutta sen pa-

```
lahin_esimies :: String -> [(String, String)] -> String
lahin_esimies tyontekija tietokanta =
  if tyontekija `elem` (keys tietokanta)
    then etsi tyontekija tietokanta
    else "ei loytynyt"

esimiehen_esimies :: String -> [(String, String)] -> String
esimiehen_esimies tyontekija tietokanta =
  if tyontekija `elem` (keys tietokanta)
    then let lesimies = etsi tyontekija tietokanta
         in if lesimies `elem` (keys tietokanta)
              then etsi lesimies tietokanta
              else "ei loytynyt"
    else "ei loytynyt"

lahin_esimies_monad :: String -> [(String, String)] -> Maybe
   String
lahin_esimies_monad tyontekija tietokanta = lookup tyontekija
   tietokanta

esimiehen_esimies_monad :: String -> [(String, String)] -> Maybe
   String
esimiehen_esimies_monad tyontekija tietokanta =
  lookup tyontekija tietokanta >>= \ lesimies ->
    lookup lesimies tietokanta
```

Kuva 4.3. *Esimiesten etsiminen Haskellissa*

lauttama arvo on maybe-monadissa. Onnistuneesti löytynyt henkilö palautetaan muodossa $Just\ henkilo$ ja epäonnistuminen ilmaistaan paluuarvolla $Nothing$. Käyttämällä tätä funktiota ja maybe-monadin tarjoamaa funktiota $>>=$ saadaan useamman esimiehen etsintä toteutettua ilman, että jokaisessa vaiheessa tarvitsee eksplisiittisesti mainita, mitä tehdään virheen sattuessa. Näin ohjelmoija pystyy paremmin ilmaisemaan koodissa sellaista toiminnallisuutta, joka ei välttämättä ole triviaalia.

Jos tarkastellaan luvussa 3 esitettyä intuitiota, jonka mukaan monadinen arvo koostuu perinteisestä arvosta ja kontekstista, voidaan Maybe-monadin kontekstia ajatella mahdollisena peitteenä annetulle arvolle. Jos arvoa ei ole peitetty, sitä voidaan käyttää muualla ohjelmassa ongelmitta. Jos taas arvo on peitetty, siihen ei päästä enää mitenkään käsiksi. Mikäli jonkin funktion yhtenä lähtöarvona on tällainen peitetty arvo, se ei voi suorittaa toimintaansa onnistuneesti, jolloin senkin täytyy palauttaa peitetty arvo.

4.2 Writer

Writer-monadi on toinen selkeä esimerkki siitä, miten monadisella rajapinnalla voidaan yhdistää puhtaita funktioita, jotka toteuttavat alunperin epäpuhdasta toiminnallisuutta. Tässä tapauksessa epäpuhtaus ilmenee siten, että funktio vaikuttaa muun ohjelman toimintaan muutenkin kuin palauttamansa arvon kautta. Kun edellä mainittu epäpuhdas toiminnallisuus halutaan toteutettaa puhtailla funktioilla, täytyy annetun funktion lähtöarvoihin lisätä sen kutsuhetkellä olemassa oleva lokiviesti ja palautettavaan arvoon vuorostaan päivitetty lokiviesti. Writer-tietotyypin ja sen monadisen rajapinnan toteutus on esitetty koodinpätkässä 4.4.

```
newtype Writer w a = Writer { runWriter :: (a,w) }

instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x, v)) >>= f = let (Writer (y, v')) = f x
                            in Writer (y, v `mappend` v')
```

Kuva 4.4. *Writer-tietotyypin ja sen monadisen rajapinnan toteutus*
[9]

Kenties helpoimmin ymmärrettävä esimerkki Writer-monadin käytöstä on lokiviestin kirjoittaminen ohjelman ajon yhteydessä. Tällöin Writer-tietotyypin parametri w on kirjoitettava lokiviesti ja a on vastaavan perinteisen funktion lähtöarvo. Kuten monadisen rajapinnan toteutuksesta huomataan, käytettävän lokiviestin tietotyypin on toteutettava monoidinen rajapinta. Tämä tarkoittaa sitä, että kyseisen tietotyypin eri arvoja voidaan yhdistää toisiinsa (mappend). Lisäksi täytyy olla olemassa sellainen arvo (mempty), jonka yhdistäminen mihin tahansa toiseen arvoon ei muuta tätä toista arvoa. Jos esimerkiksi lokiviestinä käytetään merkkijonoa, yhdistysoperaatio on $++$ ja identiteettiarvona on tyhjä merkkijono.

Yksi esimerkki edellä mainitusta toiminnallisuudesta C++ -ohjelmointikielellä toteutettuna on esitetty koodinpätkässä 4.5.

Myös koodinpätkässä 4.6 kirjoitetaan lokiviesti. Tässä toteutuksessa kirjoitettu lokiviesti vain on lisätty funktion palauttamaan arvoon.

4.3 List

Alla olevassa koodinpätkässä 4.7 esitellään listojen monadisen rajapinnan toteutus.

List-monadia käytetään esimerkiksi, kun luodaan uusi lista joidenkin olemassa olevien tietorakenteiden perusteella. Tällöin määritellään, minkälaisista alkioista uusi lista koostuu. Alla olevassa esimerkissä 4.8 luodaan kahdesta olemassa olevasta listasta uusi lista,

```
using std::string;

string log_message = ...;

int plus_two(int value)
{
    log_message += "Called plus_two";
    int new_value = value + 2;
    return new_value
}
```

Kuva 4.5. *Lokiviestin kirjoittaminen funktiokutsun yhteydessä C++ -ohjelmointikielellä*

```
plus_two :: Int -> Writer String Int
plus_two value = Writer (value + 2, "Called plus_two")
```

Kuva 4.6. *Lokiviestin kirjoittaminen funktiokutsun yhteydessä Haskellissa*

joka sisältää kaikki alkuperäisten listojen alkioista muodostetut parit.

List-monadin tapauksessa monadinen konteksti muodostuu siitä mahdollisuudesta, että sisällytettyjen arvojen määrästä ei ole mitään takeita.

4.4 State

Alla olevassa koodinpätkässä esitellään sekä State-tietotyypin että sen monadisen rajapinnan toteutus.

```
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
    return val = State $ \ s -> (val, s)
    state >>= f = State $ \ s ->
        let (a, s') = runState state s
        in runState (f a) s'
```

[10]

State-monadia käytetään etenkin silloin, kun halutaan generoida useita arvoja generaattorilla, jonka tila muuttuu jokaisen kutsun jälkeen. Yksinkertainen esimerkki tästä on laskuri, jonka arvo kasvaa yhdellä jokaisen kutsun jälkeen. Toinen esimerkki voisi olla vaikka satunnaislukugeneraattori. Vaikka satunnaislukugeneraattorin tapauksessa uuden generaattorin arvon tuottava funktio on monimutkaisempi kuin laskurin tapauksessa, molempien toiminta on päällisin puolin samanlaista. Generaattorista tuotetaan jokin uusi arvo ja

```
instance Monad List where
    return x = [x]
    x >>= f = (concat . fmap f) x
```

[6]

Kuva 4.7. *Listojen monadisen rajapinnan toteutus*

```
[(x,y) | x <- [1,2], y <- [3,4]]
-- on evaluoituna seuraava
[(1,3),(1,4),(2,3),(2,4)]
```

Kuva 4.8. *Esimerkki listan luomisesta määrittelemällä sen sisältämät alkiot*

samalla itse generaattoria muutetaan seuraavaa käyttökertaa varten. Nimenomaan tämä samankaltainen toiminta on abstrahoituna State-monadissa.

State-monadin konteksti käsittää funktiokutsusta toiseen säilyvän tietorakenteen. Tätä tietorakennetta voidaan myös muokata funktiokutsun yhteydessä.

4.5 IO

Alla olevassa koodinpätkässä esitellään yksinkertainen ohjelma, joka käyttää IO-operaatioita. Tässä tapauksessa kyse on tekstin lisäämisestä näytölle sekä käyttäjän syötteen hyödyntämisestä osana ohjelmaa.

```
main :: IO ()
main = do
    putStrLn "Hei, mika on sinun nimesi?"
    nimi <- getLine
    putStrLn ("Hauska tutustua, " ++ nimi ++ "!")
```

Yllä olevassa koodissa on käytetty do-notaatiota, joka muistuttaa imperatiivisen ohjelmoinnin rakenteita. Se on muutettavissa ilmaukseksi, joka käyttää bind-operaatiota ja lambdoja. Tässä tapauksessa saadaan alla oleva koodinpätkä.

```
main :: IO ()
main = putStrLn "Hei, mika on sinun nimesi?"
    >>= \ _ -> getLine
    >>= \ nimi -> putStrLn ("Hauska tutustua, " ++ nimi ++ "!")
```

[6]

Koska putStrLn-funktio ei tuota ohjelman kannalta hyödyllistä paluuarvoa, kyseistä paluuarvoa ei myöskään sidota mihinkään muuttujaan. Koodissa tämä ilmenee alaviivana.

5. PERUSTEISTA ETEENPÄIN

Tämä luku käsittelee kahta erilaista tilannetta, joissa monadeja on käytetty monimutkaisempien rakenteiden pohjana. Ensimmäinen näistä mahdollistaa ominaisuuksien lisäämisen jo olemassa oleviin monadeihin. Jälkimmäinen taas laajentaa koko monadien käsitettä.

5.1 Muokkaajat

Usein halutaan käyttää useampia luvussa 4 mainittujen monadien ominaisuuksia yhtä aikaa. Tämän takia yleisimmin käytetyistä monadeista on olemassa myös monadien muokkaaja. Se on muutoin samanlainen kuin monadi, jonka ominaisuuksia halutaan hyödyntää, mutta se ottaa yhtenä parametrina muokattavan monadin ja lisää halutut ominaisuudet sen päälle.

Monadien muokkaajat ovat myös itsessään monadeja. Tästä syystä useampaa erilaista monadin muokkaajaa voidaan käyttää samaan aikaan haluttujen ominaisuuksien aikaansaamiseksi.

5.2 Nuolet

Nuolet ovat yleistys monadeista. Siinä missä monadeita voi kuvitella yksittäisinä arvoina jossakin annetussa kontekstissa, nuolia voi pitää funktioina jossakin kontekstissa. [11]

Motivaatio niiden kehittämiselle lähti siitä havainnosta, että monadisten parsimisfunktioiden yhdistäminen ei onnistu ilman kyseisten funktioiden kutsumista. Parsittaessa tämä on ongelmallista, koska se kasvattaa käytetyn muistin määrää. Jos ensimmäisenä kokeiltava parsimisfunktio onnistuu parsimaan suuren osan annetusta syötteestä, mutta lopulta kuitenkin epäonnistuu koko syötteen parsimisessa, muistissa täytyy säilyttää sekä alkuperäistä syötettä, että osittain parsittua tulosta. Mitä useampia parsimisfunktioita yhdistetään, sitä useammin tämä ongelma kertautuu. Siksi olisi kätevää, että näitä funktioita voisi yhdistää toisiinsa jo ennen kuin niitä on kutsuttu. Nuolet tarjoavat tämän mahdollisuuden. [11]

6. YHTEENVETO

Tässä työssä tutkittiin monadeita ensin kategoriateorian kannalta, jossa ne on myös alun perin määritelty. Sen jälkeen tarkasteltiin, miten matemaattisesta määritelmästä päästään ohjelmointikielen toteutukseen. Lisäksi käytiin läpi muutamia käytännön esimerkkejä monadeista sekä tarkasteltiin monadien muokkaajia ja nuolia.

Työssä todettiin, että monadi on eräänlainen konteksti annetun arvon ympärillä. Kaksi tällaista kontekstia voidaan yhdistää toisiinsa assosiatiivisella operaatiolla, jota kutsutaan kategoriateoriassa nimellä μ ja ohjelmoinnissa nimellä $join$. On myös olemassa triviaali tapa tuottaa halutunlainen konteksti mistä tahansa annetusta perinteisestä arvosta. Kategoriateoriassa tätä kutsutaan nimellä η ja ohjelmoinnissa nimellä $return$. Haskellissa käytetään kuitenkin monadin määrittelyssä funktion $join$ sijasta funktiota $>>=$, jonka toteutus sisältää monadisten kontekstien yhdistämisen.

Kun ohjelmoija määrittelee haluamansa kontekstin sekä edellä mainittujen operaatioiden toteutuksen, koodia saadaan kirjoitettua tiiviimmin ja siinä keskitytään selkeämmin ilmaisemaan ei-triviaalia toimintaa.

LÄHTEET

[1] Mueller, J. P. *Functional programming*. eng. 1st edition. Hoboken, New Jersey: For Dummies, 2019. ISBN: 1-119-52749-X.

[2] Milewski, B. *Category Theory 10.1: Monads*. YouTube. 2016. URL: https://www.youtube.com/watch?v=gHiyzctYqZ0.

[3] Grandis, M. *Category theory and applications : a textbook for beginners*. eng. New Jersey, 2018.

[4] Rivas, E., Jaskelioff, M. ja Schrijvers, T. A unified view of monadic and applicative non-determinism. eng. *Science of computer programming* 152 (2018), s. 70–98. ISSN: 0167-6423.

[5] Petricek, T. What we talk about when we talk about monads. eng. *The Art, Science, and Engineering of Programming* 2.3 (2018). ISSN: 2473-7321.

[6] Klinger, S. *The Haskell Programmer's Guide to the IO Monad: Don't Panic*. eng. Centre for Telematics ja Information Technology (CTIT), 2005.

[7] Jones, S. P. Haskell 98 language and libraries: The revised report. eng. *Journal of functional programming* 13.1 (2003). ISSN: 0956-7968.

[8] Sajanikar, Y. *Haskell cookbook : build functional applications using Monads, Applicatives, and Functors*. eng. 1st edition. Birmingham, [England] ; Packt, 2017. ISBN: 9781786462657.

[9] Lipovača, M. *Learn you a Haskell for great good! a beginner's guide*. eng. 1st edition. San Francisco, Calif: No Starch Press, 2011. ISBN: 1-59327-295-2.

[10] *Hackage: The Haskell Package Repository*. URL: https://hackage.haskell.org/package/containers-0.6.4.1/docs/src/Utils.Containers.Internal.State.html.

[11] Hughes, J. Generalising monads to arrows. eng. *Science of computer programming* 37.1 (2000), s. 67–111. ISSN: 0167-6423.