Tommi Tahvanainen

Funktionaalinen ohjelmointi web-kehityksessä

Metropolia Ammattikorkeakoulu

Tieto- ja viestintätekniikan tutkinto-ohjelma

Menetelmäopinnot, Pelisovellukset

Tutkielma

Joulukuu 2022

Sisällys

1 Johdanto

Tutkielman tarkoituksena oli esitellä funktionaalisen ohjelmoinnin paradigma ja sen käyttöä modernien web-teknologioiden kanssa selvittäen syitä funktionaalisten tekniikoiden käytölle palvelimien rakentamisessa web-alustoille.

Tutkielmassa verrattiin proseduraalisen, oliokeskeisen sekä funktionaalisen ohjelmoinnin eroja eristettyjen testiesimerkkien kautta. Testiesimerkit rakennettiin tutkielmaa varten havainnolistamaan eri tekniikoiden eroja.

Tutkielmassa tarkasteltiin rekrytointitehtävänä rakennettua palvelinta, jossa käytettiin funktionaalisia ohjelmointitekniikoita. Palvelimen tarkoitus on toimia taustajärjestelmänä selainpohjaiselle sovellukselle, jolla käyttäjä voi lisätä ja poistaa kirjoja henkilökohtaiseen kirjastoonsa.

2 Funktionaalisen ohjelmoinnin historia

Funktionaalisen ohjelmoinnin edeltäjänä pidetään normaalisti lambdakalkyyliä, joka on Turing-täydellinen laskennan malli, eli sillä voidaan ilmaista mitä tahansa matemaattisia laskennan ongelmia. Lambdakalkyylille ominaista on sen deklaratiivisuus, eli operaatioita ei järjestetä Turing-laitteelle tai perinteiselle laskennalle tyypillisellä vaihekaavalla, vaan jokainen operaatio voidaan purkaa binääripuuksi. Ongelmaa siis kuvataan tietojenkäsittelytieteen näkökulmasta tietorakenteena.

Tietojenkäsittelytiede jakautui jo alkuvaiheissaan useaan koulukuntaan, joista tutkielmalle relevantteina voi pitää lambdakalkyyliä suosivaan kuntaa ja Alan Turingin seuraajia. Turingin hengessä ajattelevien mielestä tietokoneiden ja sittemmin ohjelmien tulisi mallintaa fyysisiä laitteita ja maailmaa lambdakalkyylin sijaan. Tästä maailmaamallintavasta tavasta tulisi myöhemmin myös tietojenkäsittelytieteen ja ohjelmoinnin kaupallisen kehityksen perusta. Lambdakalkyylia suosivan koulukunnan työn seurauksena funktionaalinen

ohjelmointi on edelleen käytössä tietojenkäsittelytieteen koulukunnassa esimerkiksi ohjelmoinnin ja ohjelmointikielien teorian tutkimuksessa, kuin myös kaupallisessa ohjelmoinnissa.

Ensimmäinen lambdakalkyylin toteuttava ohjelmointikieli on vuonna 1958 alkunsa saanut Lisp (List Processing), joka on vuosien varrella muodostunut kokonaiseksi kielikunnaksi. Nykypäivänä aktiivisessa käytössä on monia eri Lisp-dialekteja, joista tunnetuimmat ovat Common Lisp, Emacs Lisp ja Clojure. Lispin sekä muiden funktionaalisten kielien innoittamana funktionaalisen ohjelmoinnin konsepteja, kuten korkean tason funktiot, on myös otettu käyttöön proseduraalisiin ja olio-orientoituneisiin kieliin kuten Javascriptiin sekä Typescriptiin.

2.1 Lisp ja homoikonisuus (Homoiconicity)

Lisp:n eri dialektit toteuttavat homoikonisuuden, eli dialektit kohtelevat koodia datana ja päinvastoin. Homoikonisuutta kuvaa hyvin koodin rakenne, jonka voi purkaa binääripuuhun, joka on yleinen tietorakenne.

2.2 Javascript omaksuu funktionaalisia tekniikoita

Tutkielmassa tarkasteltavissa esimerkeissä ja rekrytointitehtävässä käytettiin Javascript-ohjelmointikieltä. Javascriptin kuvailee ECMAScript standardi, josta on julkaistu monta versiota vuodesta 1997 lähtien. Vaikka Javascript on pohjimmiltaan proseduraalinen ja imperatiivinen kieli, siihen on vuosien saatossa lisätty funktionaalisia piirteitä joiden seurauksena käyttö on poikennut enemmän funktionaaliseen ja deklaratiiviseen suuntaan.

3 Määritelmät

Funktionaalisessa ohjelmoinnissa on sääntöjä, jotka varmistavat ohjelman toiminnan tavalla, josta funktionaalisen ohjelmoinnin mahdolliset hyödyt saadaan irti. Tutkielmaa varten kiteytettiin säännöt viiteen pääsääntöön;

muuttumattomuus, funktioiden ensiluokkaisuus, tilattomuus, sivuvaikutuksettomuus sekä laskennallinen loogisuus. Lisäksi määriteltiin deklaratiivisuus sekä puhtaat funktiot, jotka ovat myös tärkeitä määritelmiä funktionaalisessa ohjelmoinnissa.

3.1 Muuttumattomuus

Muuttumattomuudella tarkoitetaan sitä, että ohjelman sisällä luotuja olioita tai muuttujia tai niiden tilaa ei muuteta luonnin jälkeen. Muuttumattomuuden takia funktionaalisessa ohjelmoinnissa luodaan usein kopioita olioista ja muuttujista, jotta alkuperäinen data säilyy koskemattomana muistissa. Muuttumattomuudella vältetään epäselvyyttä siitä, mikä olion/muuttujan tila tai sisältö on milläkin hetkellä ohjelman kulkiessa, sillä epäselvyys voi johtaa odottamattomiin lopputuloksiin ja vaikeuttaa vian löytämistä.

3.2 Ensiluokkaiset funktiot (First Class functions)

Ensiluokkaiset funktiot tai korkeamman tason funktiot toimivat funktionaalisen ohjelman perusyksikköinä. Ensiluokkaisuus tulee siitä, että funktioita kohdellaan kuten muuttujia ja olioita; funktioita voi käyttää parametreina toisille funktioille ja niitä voidaan palauttaa arvoina funktio-operaatioista. Ensiluokkaisten funktioiden käyttöä kuvaa hyvin aiemmin mainittu binääripuu, jossa operaatiot ja arvot jaetaan tietorakenteeseen.

3.3 Tilattomuus

Kuten muuttumattomuus, tilattomuus liittyy ohjelman suorituksen aikana tapahtuviin operaatioihin ja niiden kohteina oleviin muuttujiin ja olioihin. Tilattomuudella tarkoitetaan että ohjelman ei tulisi ylläpitää minkäänlaista olio-tilaa suorituksen aikana. Tällöin jokainen funktio operoi eristyksissä muista ohjelman funktioista ja on luotettavampi sekä jokaisen funktion uudelleenkäytettävyys säilyy, koska ne eivät ole riippuvaisia minkään olion tilasta.

3.4 Sivuvaikutuksettomuus

Funktionaalisissa ohjelmissa fuenktioiden ei tulisi aiheuttaa sivuvaikutuksia. Sivuvaikutuksettomuus useasti muodostuu luonnostaan jos muuttumattomuuden ja tilattomuuden sääntöjä noudatetaan, sillä tyypillisimmät sivuvaikutukset ovat muutoksia funktion ulkopuoliseen tilaan tai muutoksia viittauksella muistiin kun ”muuttuva” olio on asetettu funktion parametriksi.

Syöttö- sekä ulostulo-operaatiot ovat myös sivuvaikutuksia. Täydellistä sivuvaikutuksettomuutta on täten vaikea säilyttää kielissä, joissa ei ole implementoitu jotain erillistä tapeaa hoitaa ohjelman vuorovaikutusta ulkoisen maailman kanssa. Haskell on esimerkki ohjelmointikielestä, jossa on implementoitu mekanismi nimeltä ”monadi”, jolla tämänkaltainen vuorovaikutus säilyy sivuvaikuksettomana.

3.5 Laskennallinen loogisuus

Funktionaalisessa ohjelmoinnissa pyritään siihen, että jokainen funktio toimisi laskennallisesti johdonmukaisesti; jos kerroin funktiolle annetaan parametreina kokonaisluku 2 ja muuttuja X tulisi ulostulon olla aina 2*X. Tätä logiikkaa sovelletaan monimutkaisempiin operaatioihin, joissa esimerkiksi iteroidaan isoa tietorakennetta.

3.6 Puhtaat funktiot

Kun funktio täyttää sivuvaikutuksettomuuden sekä laskennallisen loogisuuden säännöt, kyseessä on puhdas funktio. Puhtaiden funktioiden käyttö ensisijaisesti ohjelman perusyksikköinä tilattomuuden ja muuttumattomuuden kanssa johtaa täysin funktionaaliseen ohjelmaan.

3.7 Deklaratiivisuus

Funktionaalinen koodi on deklaratiivista, missä proseduraalinen koodi on imperatiivista. Web-ohjelmointi on useasti sekoitus molempia ohjelmointiparadigmoja.

Deklaratiiviselle ohjelmoinnille on tyypillistä, että ohjelman ja aliohjelmien lopputulosta kuvaillaan, kun taas imperatiivisessa ohjelmoinnissa keskitytään enemmän ohjelman kulun kuvailuun tai prosessin välivaiheiden määrittelyyn. Vastakkainasettelua paradigmojen välillä kuvaillaan monesti miten/mitä-asettelulla, jossa imperatiivinen keskittyy siihen miten jotain tehdään ja deklaratiivinen siihen mitä tehdään.

4 Ohjelmakoodiesimerkkejä

4.1 Korkeamman tason funktiot

```
const toMax = (max, value) => Math.Max(max, value);
[1,2,3,4].reduce(toMax);
```

Esimerkkikoodi 1. Javascript ohjelma, jossa on määritelty funktio toMax(), joka palauttaa korkeimman arvon siihen sijoitetuista arvoista. Ohjelma on esimerkki funktioiden sijoittamisesta muuttujaan, ja kuinka funktioita voidaan käyttää parametreina toisille funktioille, tässä tapauksessa Array.reduce():lle.

4.2 Korkeamman tason funktioiden ja proseduraalisten toistorakenteiden vertailu

```
function getPrimes(array) {
     let primes = [];
     let flag = true;

     for (number in array) {
          flag = true;
          if (number == 0 || number == 1) flag = false;
          else if (number > 1) {
               for (let i = 2; i <= number / 2; ++i) {
                    if (number % i == 0) {
                         flag = false;
                         break;
                    }
               }
          }
          if (flag == true) primes.push(number);
     }
     console.log(`Prime numbers: ${primes}`);
     return primes;
}
```

Esimerkkikoodi 2. Proseduraalinen Javascript aliohjelma, joka tulostaa ja palauttaa listan alkulukuja sille parametrina annetusta listasta. Funktio tarkistaa listan luvut proseduraalisesti, hyödyntäen perinteisiä toisto- ja valintarakenteita.

```
const isPrime = n =>
     n <= 1
          ? false
          : !Array.from(new Array(n), (el,i) => i + 1)
               .filter(x => x > 1 && x < n)
               .find(x => n % x === 0);

const array = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17];
const primes = array.filter(isPrime);
```

Esimerkkikoodi 3. Funktionaalinen Javascript ohjelma, jossa on määritelty aliohjelma isPrime() joka määrittelee onko sille parametrina annettu luku alkuluku. Aliohjelma sijoitetaan Array.filter():lle parametrina, joka käydessään alkuperäistä listaa läpi tarkistaa onko kyseessä alkuluku isPrime():ssa määritellyllä tavalla ja palauttaa alkion uuteen listaan mikäli näin on.

Verrattaessa proseduraalista ja funktionaalista esimerkkiä huomataan proseduraalisen esimerkin sisältävän enemmän sisennystä sekä kontekstirajoja, joka tekee ohjelman lukemisesta sekä ymmärtämisestä

haastavaa. Funktionaalisessa esimerkissä käytetty valintarakenne (eng. ternary operator) sekä korkeamman tason funktioiden hyödyntäminen kuvailee tarkemmin mitä aliohjelma isPrime() tekee. Koodin luettavuus on monesti mielipidekysymys, mutta tässä tapauksessa korrektiuden kannalta deklaratiivisuus tuo selkeyttä koodiin.

Esimerkkikoodien 2. ja 3. välillä ei ole merkittävää eroa suoritusnopeudessa ellei alkioita ole todella suuria määriä. Tapauksissa, joissa tiedetään ettei funktiota käytetä suurien tietorakenteiden iterointiin, voidaan valita kehittäjille mielekkäämpi ja selkeämpi tapa toteuttaa funktio.

4.3 Kirjastopalvelin

Kirjastopalvelin kirjoitettiin rekrytointitehtävänä ja ammentaa sekä proseduraalisia että funktionaalisia tekniikoita.

```
export const dbPromise = async () => open({
     filename: ’books.db’,
     driver: sqlite3.Database
});

const queryRun = async (sql: string, params: SqlParams) => {
     const promise = await dbPromise();
     return await promise.run(sql, params);
};
```

Esimerkkikoodi 4. Osa laajempaa Typescript-moduulia, jossa aliohjelmat dbPromise() ja queryRun() muodostavat yhteyden tietokantaan ja ajavat esimerkkikoodissa 5. määriteltyä palvelinkoodia tietokantaa vasten. Moduulissa käytettiin ”semi-funktionaalista” koodia, jossa funktiot ovat sivuvaikutuksettomia ja kutsuvat muita funktioita, mutta eivät ole täysin puhtaita funktioita.

```
const create = async (book: SqlParams) => {
     const sql = `insert into
     books(title,author,year,publisher,description) values
     (?,?,?,?,?)`;
     return await db.queryRun(sql, book);
};

const searchAll = async (search: SqlParams) => {
     const sql = buildSearchQuery(search);
     return await db.queryAll(sql, search);
};

export const buildSearchQuery = (search: SqlParams) => {
     let sql = `select * from books where `;
     let first = true;
     if (search[0]) {
          sql = sql.concat(`author = ? `);
          first = false;
     }
     if (search[1]) {
          if (first) {
               sql = sql.concat(`year = ? `);
               first = false;
          }
          else sql = sql.concat(`and year = ?`);
     }
     if (search[2]) {
          if (first)
               sql = sql.concat(`publisher = ? `);
          else sql = sql.concat(`and publisher = ?`);
     }
     return sql;
}
```

Esimerkkikoodi 5. Osa laajempaa Typescript-moduulia, jossa funktiot create() ja searchAll() muodostavat SQL-lausekkeita funktioiden parametrien perusteella ja sijoittavat SQL-lausekkeet tietokantamoduulin funktioihin. Moduulia varten muodostettiin apufunktio buildSearchQuery(), joka rakentaa SQL-lausekkeen parametreistaan riippuen. Apufunktiossa hyödynnettiin proseduraalista ohjelmointia, sillä merkkijonojen ketjutus parametrejen olemassaolosta riippuen osoittautui hankalaksi puhtaasti funktionaalisella ratkaisulla.

5 Yhteenveto

Lähteet

Käytä jompaakumpaa alla olevista viittausjärjestelmistä. Poista se, jota et aio käyttää.

Harvard-järjestelmä (nimi-vuosijärjestelmä):

Lisää lähteet aakkosjärjestyksessä.

Aaltonen, Pietu. 2019. Tutkiva kirjoittaja ammattikorkeakoulussa. Opinnäytetyö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

Oppivainen, Sanelma. 2020. Opinnäytetyön raportointiopas. Helsinki: Kaarikustantamo.

Vancouver-järjestelmä (numeroviitejärjestelmä):

Lisää lähteet siinä järjestyksessä, kuin ne on mainittu tekstissä.

1 Oppivainen, Sanelma. 2020. Opinnäytetyön raportointiopas. Helsinki: Kaarikustantamo.

2 Aaltonen, Pietu. 2019. Tutkiva kirjoittaja ammattikorkeakoulussa. Opinnäytetyö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.