haskell-go

Ohjelmointikielet ja paradigmat

Harjoitustyöksi toteutimme Go lautapelin. Toiminnallisuus (Back end) on toteutettu Haskell ohjelmointikielellä, jonka lisäksi teimme web käyttöliittymän Javascriptilla käyttäen React viitekehystä.

Funktionaalinen paradigma

Paradigman esittely

Funktinaalisuudella tarkoitetaan yleensä sitä, että funktioden käsittely on saman kaltaista kuin muidenkin tietorakenteiden. Funktioita voidaan antaa muihin funktioihin argumenteiksi ja niitä voidaan palauttaa kutsun lopuksi. Funktiota voidaan yleensä myös tallentaa esimerkiksi tietokantoihin.

Funktionaalisuus mahdollistaa korkeampiasteisten funktioiden luomisen. Korkeamman asteen funktio on funktio, jonka argumenttina on myös funktio. Korkeamman asteen funktiot voivat helpoittaa ohjelmointia ja mahdollistavat siistimpää koodia. Esimerkiksi mappaus ja suodatus ovat yleisiä korkeamman asteen funktioita. Seuraava esimerkki on Javascriptiä.

```
function tuplaa(numero) = { return numero * 2 }

function suodata(numero) = { return numero < 5 }

var lista = [2,6,1,5,7,9,0,4]
var tuplat = lista.map(tuplaa)
var suodatetut = lista.filter(suodata)
```

Ilman korkeamman asteen funktioita sama toiminnallisuus täytyisi toteuttaa toistorakenteilla.

```
var lista = [2,6,1,5,7,9,0,4]

var tuplat = []
for (var i = 0; i < lista.length; i++) {
   tuplat.push(tuplaa(lista[i]))
}

var suodatetut = []
for (var i = 0; i < lista.length; i++) {
   if (suodata(lista[i])) {
      suodatetut.push(lista[i])
   }
}
```

Monissa kielissä on myös mahdollista määrittää lambda funktioita, jotka yleensä tarkoittavat funktioita, joita ei ole sidottu mihinkään nimeen. Lambda funktioita voidaan käyttää esimerkiksi juuri mappausten tai suodatusten kanssa.

```
var lista = [2,6,1,5,7,9,0,4]
var uudet = lista.map((i) => {
   return Math.sin(i*30) * 5 - 2
})
```

Puhtaus

Funktionaalisen kielen puhtaudella tarkoitetaan, että funktioilla ei ole sivuvaikutuksia. Sivuvaikutukset tarkoittavat sitä, että funktiot muuttavat ohjelman tilaa oman suorituksen aikana. Funktiolla voi kuitenkin olla omaa muistia ja ne voivat palauttaa jotain. Jos käytössä on ainoastaan puhtaita funktioita, tietorakenteet eivät voi mutatoitua vaan funktiot palauttavat aina uuden rakenteen.

Ohjelmointikielen puhtaus helpottaa ohjelman tilan hallintaa. Jos funktiot eivät ole puhtaita, voivat ne muokata ohjelman tilaa, jolloin tila riippuu hyvin paljon kontrollivuosta ja siitä, missä järjestyksessä funktioita kutsutaan. Myös ohjelmoidessa voi olla rasittavaa, jos ei tiedä millä tavalla funktio muuttaa ohjelman tilaa. Esimerkiksi Javascriptissä listoille tehtävien operaatioiden käyttäytymistä voi olla vaikea muistaa.

```
let lista = [2,6,1,5,7,9,0,4]
lista.map(i => i*2) // ei muuta listaa, vaan palauttaa mapatun listan
lista.sort((a, b) => { return a < b }) // muuttaa alkuperäistä listaa
```

Puhtaiden funktioiden käyttäytymisen helpompi ennustaminen johtaa vähempään määrään virheitä.

Jos käytetään ainoastaan puhtaita funktioita, ohjelman tilaa täytyy hahmottaa eri tavalla kuin perinteisillä epäpuhtailla kielillä. Koska funktiot eivät voi muuttaa ohjelman globaalia tilaa, ohjelman suorituksen tila riippuu ainoastaa parasta aikaa suoritettavista funktioista, niiden sisäisestä tilasta ja suorituskohdasta.

Puhtauden lisäksi funktiot ovat yleensä ainoastaan yhden parametrin funktioita, jotka palauttavat aina jotain. Tällöin kahden parametrin funktiot koostuvat tavallaan kahdesta funktiosta. Ylimpänä on funktio, joka ottaa parametrin ensimmäisen parametrin, ja palauttaa funktion, joka ottaa parametrikseen toisen parametrin ja palauttaa tuloksen. Funktion saama parametri on sidottu kyseisen funktion sisällä, jolloin se on sidottu myös sen palauttaman funktion sisällä. Kun usean parametrin funktioita kutsutaan, sidotaan siis parametreja niin paljon kun niitä annettiin, ja joko funktio jos parametreja on vielä jäljellä, tai tulos.

Tyypillinen ohjelman rakenne

Puhtailla funktionaalisilla kielillä ohjelmointi on hyvin kuvailevaa. Funktioita luodaan käyttämällä aiempia funktioita. Yleensä lopulta päädytään yhteen ylimpään funktioon, joka on lopulta koko ohjelman suoritus. Kontrollivuohon ei kiinnitetä huomiota vaan se seuraa implisiittisesti kuvatuista funktioista. Perinteisiä toistorakenteita ei voida käyttää, sillä ne vaatisivat globaalin tilan muuttamista joka kierroksella. Toisto täytyykin toteuttaa rekursiolla, jossa funktio kutsuu itseään uudella syötteellä.

Yleensä käytetään paljon mappauksia, suodatuksia tai taittoja. Pohjimmillaan nämäkin on kuitenkin toteutettu rekursiolla. Taitot ovat listan (tai muun tietorakenteen) yli tehtäviä operaatioita, jossa annettu funktio suoritetaan ensin jollekkin alkusyötteelle ja yhdelle alkiolle, ja tämän jälkeen saatu tulos syötetään takaisin funktioon uuden alkion kanssa. Taitto ikäänkuin kerryttää tulosta tietorakenteen yli suorittaessa alkioittain.

Funktioille voi myös määrittää eri sisältö niiden saaman syötteen perusteella. Esimerkiksi Haskelissa.

```
tuplaaJosViisi 5 = 5*2
tuplaaJosViisi i = i
```

IF/ELSE kontrollirakenteiden sijaan käytetään juuri edellistä syötteen sovitusta tai esimerkiksi case operaattoria jolla voidaan eri tilanteille määritellä eri tulos.

Tyypitys

Funktionaaliset kielet ovat usein vahvasti tyypitettyjä. Tämä tarkoittaa yleensä staattista käännösaikana tarkistettavaa tyypitystä. Kaikilla funktioille ja arvoille voidaan antaa tyyppi, ja jos tyyppiä ei ole annettu, yrittää kääntäjä itse määrittää oikean tyypin. Koska funktiot ovat puhtaita, ei ole tarvetta mutatoituville tietorakenteille ja dynaamiselle tyyppisidonnalle. Staattinen tyypitys parantaa ohjelman virheettömyyttä, sillä tyypitysvirheet huomataan jo käännösaikana.

Usein voidaan määrittää omia algebraalisia datatyyppejä, joiden arvoalueet voidaan määrittää itse, ja uusia tyyppejä voidaan muodostaa yhdistämällä muita tyyppejä.

Laskentamallit

Laiska vs ahkera

Samanaikaisuus

Haskell

Haskell on puhdas funktionaalinen ohjelmointikieli, joka käyttää laiskaa laskentaa.

Haskelissa voidaan muodostaa uusia algebraalisia datatyyppejä.

Back end

Moduulit

Seuraavasta taulusta näkyvät ohjelmassa käytetyt moduulit ja niiden vastuut:

Moduuli	**Vastuu**
Board.hs	Luo datatyypit Side ja PlaceData, jotka kuvaavat puolta (Valkoinen/Musta) ja yhdessä ruudussa olevaa dataa. Luo tyyppi aliaksia paikalle ja laudalle. Luo funktioita laudan anaysoinnille ja manipuloinnille.

Moduuli	Vastuu
Region.hs	Luo tyyppi alueelle (lista paikkoja) ja funktioita yhtenäisten alueiden etsimiselle, alueiden reunojen hakemiselle ja alueiden sisällön analysoimiselle ja muokkaamiselle.
GameState.hs	Luo datatyypit pelaajan datalle ja pelin tilalle.
Move.hs	Luo datatyyppi siirrolle. Luo pelilogiikka käyttäen hyväksi edellä kuvattuja moduuleita. Sisältää funktiot siirtojen oikeellisuuden tarkastukselle (moveIsValid) ja siirtojen suorittamiselle (executeMove).
GoHttpServer.hs	Luo Happstack kirjaston avulla HTTP palvelin. Muodosta HTTP rajapinta, jonka avulla peliä voidaan peliä. Pelin tila voidaan muuntaa JSON muotoiseksi ja data välitys HTTP:n yli on juurikin JSON muotoista.

Data tyypit

Ohjelmassa on määritelty seuraavat datatyypit ja tyyppialiakset laudan kuvaamiseksi.

```
data Side = Black | White
data PlaceData = Empty | Stone { side :: Side }
type Place = (Int, Int)
type Row = [PlaceData]
type Board = [Row]

data RegionType = Undefined | RegionType PlaceData deriving (Eq, Show)
type Region = [Place]
```

Lauta koostuu siis kaksiuloitteisesta listasta alkioita, joiden arvo voi olla tyhjä, valkoinen kivi tai musta kivi.

```
data Player = Player {
   playerSide :: Side,
   captured :: Int,
   hasPassed :: Bool,
   hasFinished :: Bool,
   finalScore :: Int
}

data GameState = GameState {
   board :: Board,
   prevBoard :: Board,
   playerInTurn :: Player,
   otherPlayer :: Player,
   gameOver :: Bool,
}

data MoveType = StonePlacing | Passing | Finishing
data Move = Move {
    moveType :: MoveType,
```

```
    place :: Place
}
```

HTTP palvelin

Parannettavaa

Virheenhallinta monadeilla

Reaktiivinen paradigma

Paradigman esittely

Ohjelman käyttöliittymä on tehty käyttäen reaktiivista React viitekehystä. Reaktiivisuus tarkoittaa ohjelman sisäistä reagointia tiettyihin signaaleihin. Signaalit voivat olla esimerkiksi ohjelman muuttuva tila, tai käyttäjältä saatu syöte. Paradigmana reaktiivisuus pyrkii olemaan kuvaileva ja pyrkii helpottamaan tiedon jakamista ohjelmassa, eli keskittyy tietovuohon.

Reaktiivisessa ohjelmoinnissa voidaan luoda monimutkaisia tietovoita, joilla uuden tiedon jakamisen hallitseminen helpottuu. Uusia tietovoita voidaan muodostaa esimerkiksi käyttäjän syötteisiin, raskaiden asynkroonisten toimintojen suorittamiseen tai datan hakemiseen verkon yli. Tietoa voidaan tämän jälkeen muokata erilaisilla funktioilla ja lopulta välittää se kyseisen tietovuon tilanneille tahoille.

Koska tietovuo voi kulkea useiden asynkroonisten vaiheiden läpi, jotka saattavat kestää eri pituisia aikoja, usen voiden yhteyteen on implementoitu "Back pressure" mekanismi, jolla pyritään estämään tiedon tarpeeton kasautuminen vaiheiden välisii puskureihin ja turha muistin täyttyminen. Mekanismissa vaiheet voivat esimerkiksi kertoa kuinka paljon ne ovat valmiita ottamaan uutta dataa prosessoitavaksi.

Kontrollin ja tilan hallinta

Usein ohjelman kontrollin ja tilan hallinta voi olla monimutkaista. Reaktiivinen ohjelmointi kuitenkin pyrkii vaihtamaan ohjelmoinnin kontrollin hallinnasta tietovoiden hallintaan. Tietovoiden luominen on deklaratiivista, eli koodiin kuvataan halutut rakenteen sen sijaan, että siihen kirjattaisiin imperatiivisesti toiminnot, jotka toteuttavat halutut rakenteet. Deklaratiivisesti kuvattu ohjelma on usein helpommin hallittavissa kuin imperatiivisesti kirjattu ohjelma.

Paradigmassa kontrollivuo seuraa kuvatuista rakenteista implisiittisesti. Tiedon valuessa voissa eteenpäin, kontrolli siirtyy sitä seuraavaksi tarvitsevalle taholle. Tietovuot ovat abstraktio kontrollin jakamiselle tiedon muuttuessa, ja tiedon välitykselle ohjelman sisällä.

Samanaikaisuus

Tietovuolle yleensä koostuu selvästi erillisistä tapahtumista/tietorakenteista. Sille voidaan tehdä erilaisia operaatiota. Suodatus karsii tietovuosta tapahtumia jollain perusteella. Mappauksella taas voidaan jokainen erillinen tietorakenne käsitellä jollain tapaa uudeksi tietorakenteeksi.

Reaktiivisilla tietovoilla voidaan myös asynkroonisuutta ja samanaikaista laskentaa hyödyntää helpommin. Kun tietovuot kuvataan deklaratiivisesti, operaatioiden välinen riippuvuus on huomattavasti selkeämpää, kuin

kontrollivuon monimutkaisen jakamisen kanssa. Esimerkiksi verkon yli suoritettavat haut voidaan suorittaa omassa säikeessä, ja vastauksen tullessa palauttaa tietovuon kontrolli alkuperäiseen säikeeseen. Myös esimerkiksi mappauksessa tehdyt toiminnot voidaan suorittaa samanaikaisesti.

React

Projektissa käytetty React viitekehys pyrkii irroittamaan DOMin (Document object model), eli nettisivun sisällön, joka piirretään käyttäjälle, sivun toiminnallisuudesta. Tietovuo pyritään muodostamaan tilan ja näkymän välillä niin, että tieto automaattisesti valuu tilasta näkymään, ilman että ohjelman kirjoittajan tarvitsee manipuloida näkymää. Näkymä luodaan siis implisiittisesti ohjelman tilasta. Reactilla on oma virtuaalinen DOM, jonka muutokset siirretään automaattisesti sivun varsinaiseen DOMiin. Virtuaalisen DOMin kuvaamiseksi on luotu JSX syntaksi, joka muistuttaa HTML:llän syntaksia.

React mahdollistaa helposti hallittavien yhden sivun web applikaatioiden luonnin. Monimutkaisissa yhden sivun applikaatioissa sivun tilaa voi olla yllättävän paljon. Reactin avulla sivu voidaan jakaa uudelleenkäytettäviin komponentteihin, joilla voi olla omaa tilaa. Komponenttien sisällä voi olla muita komponentteja, tai matalimmilla tasolla perinteisiä HTML elementtejä/tageja. Itseasiassa nämä komponentit muodostavat Reactin virtuaalisen DOMin. Virtuaalisten komponenttien tilan on kytketty reaktiivisesti oikean DOMiin, ja muutokset tilassa aiheuttavat muutoksia näkymään.

React käyttää edistynyttä algoritmia muutoksen analysoimiseksi. Muutokset eivät aina aiheuta kovinkaan suurta muutosta näkymään. React analysoi muutokset ja muutta oikeaa DOMia ainoastaan tarvittavan määrän.

React hoitaa ohjelman reaktiivisuuden itse, ja ohjelmoijan ei tarvitse määrittää tietovoita itse. Riittää kun ymmärtää miten viitekehys välittää tietoa ja noudattaa kehyksen määräämiä sääntöjä esimerkiksi tilan muuttamiselle. Reaktiivisuus helpottaa kuitenkin monimutkaisten web ohjelmien luomista huomattavasti. Näkymän abstraktointi virtuaaliseen DOMiin tekee ohjelman tilasta paljon hallittavampaa kuin tilan säilyttämisestä aidossa DOMissa, tai jokaisen näkymää muuttavan muutoksen explisiittistä välittämistä ohjelman tilan lisäksi näkymään.

Front end

Pelin back end oltiin toteutettu Haskelilla niin, että HTTP palvelimen kuutelee kutsuja ja jos kutsu on joko uuden pelin aloitus, tai meneillään olevan pelin tila ja seuraava siirto, palvelin vastaa seuraavalla pelin tilalla. Front endin tehtäväksi jää siis vain tilan piirtäminen, käyttäjän klikkausten kuunteleminen ja uuden siirron ja nykyisen tilan lähettäminen HTTP pyynnöllä palvelimelle.

Valitsimme front endin toteuttamisekis seilaimessa näkyvän web käyttöliittymän. Valinta tuntui helpolta, sillä palvelin toimii HTTP pyyntöjen avulla. Toteutukseen valitsimme reaktiivisen React viitekehyksen. React on erittäin suosittu ja sen tilanhallinta ja reaktiivinen renderöinti tuntui hyvältä valinnalta. Koska palvelin toimii ainoastaan HTTP pyyntöjen ylitse, ja välitetty data on JSON muotoista, valitsimme kehyksen, jolla voidaan helposti toteuttaa HTTP pyyntöjä ja saatu vastaus tallentaa ohjelman tilaan.

Komponentit

Yhden sivun applikaatio koostuu muutamasta eri komponentista. Ylimpänä olevan App komponentin tilassa on tieto siitä, ollaanko valikossa vai pelissä. Tilassa on myös uuden pelin aloittamiin tarvittava tieto laudan koosta.

MainMenu komponentti mahdollistaa laudan koon valitsemisen ja pelin aloittamisen. Käyttäjän klikkaukset nousevat App komponentin tilanmuutoksiksi.

Kun peli alkaa muutetaan App komponentin tila valikko tilasta peli tilaan. Tällöin virtuaaliseen DOMiin laitetaan komponentti Game. Game komponentti lähettää heti kiinnittymisensä jälkee pyynnön palvelimelle, jossa se pyytää uutta peliä tietyllä valitulla laudankoolla. Vastauksen saavuttua Game tallentaa saadun pelin tilan omaan tilaansa ja renderöi ActionMenu ja Board komponentit, joilla käyttäjä voi tehdä siirtoja. Board komponentti saa Game komponentin tilasta sen hetkisen laudan tilanteen jonka se renderöi. Käyttäjän valitessa siirron, Game komponentti lähettää tiedon nykyisestä pelin tilasta ja pelaajan siirrosta palvelimelle, joka vastaa uudella pelin tilalla, joka jälleen tallennetaan Game komponenttiin ja tapahtuu uudelleen renderöinti. Pelin loputtua (tieto saadaan palvelimelta) renderöidään pisteeet ja mahdollisuus siirtyä takaisin valikkoon.

Sivua luodessa pyrimme pitämään sivulla tapahtuvan pelilogiikan minimissä. Board komponentilla olvat ruudut ovat kuitenkin valittavissa siirroiksi ainoastaan, jos ne ovat tyhjiä. Tämä oli erittäin helppo implementoida. Kaikki tyhjät siirrot ovat kuitenkin valittavissa, vaikka pelilogiikka ei niitä siirroiksi sallikkaan. Tällöin palvelin vastaa lähetettyyn pyyntöön samalla pelitilalla. Kyseinen ratkaisu ei ole kaikkein informatiivisin käyttäjälle, mutta yksinkertaisuuden vuoksi se on mielestämme tarpeeksi hyvä.