RINNAKKAISUUS FUNKTIONAALISESSA OHJELMOINTIKIELESSÄ HASKELL

Anssi Tilli
Pro gradu -tutkielma
Tietojenkäsittelytiede
Itä-Suomen yliopiston
Tietojenkäsittelytieteen laitos
2012

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta
Tietojenkäsittelytiede

TILLI, A.: Rinnakkaisuus funktionaalisessa ohjelmointikielessä Haskell
Pro gradu -tutkielma, 65 s., 4 liitettä (12 s.)
Pro gradu -tutkielman ohjaaja: FT Matti Nykänen
2012

Avainsanat: Rinnakkaisuus, funktionaalinen ohjelmointi, Haskell, Glasgow Parallel Haskell, Data Parallel Haskell

Yksittäisen prosessorin nopeutta ei voida tällä hetkellä käytössä olevilla menetelmillä juurikaan nostaa, joten moniydinprosessorit ovat prosessorikehityksen tämänhetkinen suunta. Myös ohjelmistokehityksen on vastattava tähän muutokseen lisäämällä rinnakkaisuutta ohjelmiin. Rinnakkaisuus on kuitenkin tunnetusti haastavaa ja lisää huomattavasti ohjelmien virhealttiutta kun ohjelmoijan on huolehdittava normaalin ohjelmalogiikan lisäksi myös rinnakkaisuuteen liittyvistä ongelmista kuten säikeiden luonnista ja niiden kommunikoinnista. Puhtaasti funktionaaliset ohjelmointikielet kuten Haskell tarjoavat mielenkiintoisen vaihtoehdon rinnakkaisuuden toteuttamiseen, koska niissä rinnakkaisuus on automatisoitavissa helpommin. Kun rinnakkaisuuden vaatimat käytännön toimenpiteet kuten säikeiden luonti voidaan automatisoida, ohjelmoija voi jälleen keskittyä pelkästään ohjelman oman logiikan toteuttamiseen hyödyntäen silti moniydinprosessoreita ohjelman suorituksessa.

Täysin implisiittinen rinnakkaisuus ei ole vielä toteutunut edes funktionaalisissa kielissä. Tässä tutkielmassa esitellään Haskell-ohjelmointikielen kaksi lisäkirjastoa, joiden avulla suurin osa rinnakkaisuuden käytännön toteutuksesta voidaan siirtää kääntäjän ja ajoympäristön huoleksi: Glasgow Parallel Haskell ja Data Parallel Haskell. Molemmissa ohjelmoija antaa kääntäjälle ja ajoympäristölle vihjeitä missä kohtaa ohjelmaa rinnakkaisuutta kannattaisi hyödyntää, mutta lopulta ajoympäristö päättää mitä itse asiassa suoritetaan rinnakkain. Ajoympäristö huolehtii kaikista rinnakkaisuuteen liittyvistä käytännön toimenpiteistä.

Lopuksi tutkielmassa on esimerkkiohjelman avulla tutkittu minkälaista lisätyötä ohjelmoijalta vaatii pienen tavallisen Haskell-ohjelman muuntaminen rinnakkaiseksi Glasgow Parallel Haskellin avulla. Samalla tutkittiin minkälaisia parannuksia suorituskykyyn tällä muunnoksella saavutetaan.

Esipuhe

Tämä tutkielma on tehty Itä-Suomen yliopiston tietojenkäsittelytieteen laitokselle syksyn 2010 ja kevään 2012 välisenä aikana. Tutkielman ohjaajana on toiminut Matti Nykänen, jolle haluan osoittaa kiitoksen.

Kuopiossa 14.6.2012

Anssi Tilli

Käsitteet ja lyhenteet

DPH	Data Parallel Haskell
GHC	Glasgow Haskell Compiler
GHCi	Glasgow Haskell Compilerin interaktiivinen versio, jossa Haskell-funktioita ja ohjelmia voi suorittaa tulkattuina
GPH	Glasgow Parallel Haskell
Säie	Säie kuvaa itsenäisen suorituspolun prosessin sisällä, mutta se jakaa isäntäprosessinsa tilan ja muistin [Lee06]
Tehtävärinnakkaisuus	Kahden toisistaan riippumattoman tehtävän suorittaminen rinnakkain [Mei11]
Tietorinnakkaisuus	Saman operaation suoritus eri arvojoukolle rinnakkain [PCL08]
Tynkä	Suomenkielinen vastine sanalle thunk, joka tarkoittaa laiskan evaluoinnin kielessä lauseketta, jota ei ole vielä evaluoitu

SISÄLTÖ

LIITTEET

1 JOHDANTO

Mahdollisuus rinnakkaislaskentaan ohjelmointikielissä on ollut olemassa jo vuosikymmeniä. Tavallisesti sitä on hyödynnetty supertietokoneissa ja näytönohjaimissa. Panostukset rinnakkaisuuden kehittämiseen ja hyödyntämiseen tavallisissa ohjelmissa ovat olleet kuitenkin verrattain vähäiset johtuen prosessoreiden jatkuvasta nopeutumisesta. Tavallisissa ohjelmissa ei ollut kannattavaa uhrata aikaa ja vaivaa rinnakkaisuuden toteuttamiseen, koska vuoden päästä uusi prosessori hoiti tehtävän jo vähintään yhtä nopeasti, usein nopeamminkin. Rinnakkaisuuteen olivat valmiita panostamaan lähinnä asialle vihkiytyneet harrastajat ja rinnakkaisuuden asiantuntijat. Kaksituhatluvun ensimmäisen vuosikymmenen puoliväliin tultaessa prosessoreiden nopeuttaminen kellotaajuuksia nostamalla tuli kuitenkin tiensä päähän kun piin fyysiset rajat tulivat vastaan. Moniydinprosessoreista muodostui alan uusi suunta. Samalla myös rinnakkaisuudella alkoi olla merkitystä myös tavalliselle ohjelmoijalle, sillä rinnakkaisuudesta tulikin ainoa tapa nopeuttaa ohjelman suoritusta. Myös ohjelmointikielten kehittäjät ovat reagoineet tähän muutokseen, ja rinnakkaisohjelmoinnin kehitys on vilkkaampaa kuin koskaan.

Tavallisten ohjelmien kirjoittaminen voi olla jo yksinään hankalaa, mutta rinnakkaisen ohjelman toteuttaminen lisää vaikeusastetta ja ohjelmoijan työmäärää huomattavasti. Ohjelmoijan on huolehdittava tarvittavien säikeiden luonnista, synkronoinnista ja niiden välisestä kommunikoinnista. Samalla mahdollisten virhetilanteiden määrä kasvaa ja ohjelman oikeellisen toiminnan varmistaminen vaikeutuu entisestään. Ideaalitilanteessa ohjelmoija kirjoittaisi rinnakkaisen ohjelman samalla tavalla kuin perättäisenkin ohjelman ja ajoympäristö huolehtisi laskentakuorman sopivasta jakamisesta käytettävissä oleville prosessoreille. Vaikka tällainen implisiittinen (tai automaattinen) rinnakkaisuus on vielä kaukana todellisuudesta, on siihen liittyvä tutkimus aktiivista. Haskell on funktionaalinen ohjelmointikieli, jossa on jo onnistuttu siirtämään suuri osa rinnakkaisuuden vaatimasta työstä ohjelmoijan harteilta ajoympäristön huoleksi.

Nykyisin tietojenkäsittelyalan opintoja ja vielä enemmän käytännön työtä dominoivat olio-

pohjaiset kielet. Menneinä vuosikymmeninä dominoivassa asemassa olivat proseduraaliset kielet, joista C-kieli on edelleen lähes korvaamattomassa asemassa matalan tason ohjelmointia tehtäessä. Vaikka olio-ohjelmointi ja proseduraalinen ohjelmointi poikkeavat toisistaan hyvinkin paljon, on niiden imperatiivinen perusajatus kuitenkin lopulta sama: suoritetaan perättäisiä käskyjä, jotka muuttavat muistin tilaa. Suurimmalle osalle ihmisistä tämä paradigma on ohjelmoinnin synonyymi. Funktionaalisen ohjelmoinnin paradigma on hyvin erilainen, se korostaa arvojen tuottamista tilan muutosten sijaan. Funktionaalisessa ohjelmoinnissa ei tunneta perinteisiä muuttujia vaan pääosassa ovat funktiot, jotka palauttavat aina jonkin arvon. Funktionaalinen ohjelmointi on ollut olemassa lähes yhtä kauan kuin imperatiivinenkin ohjelmointi, mutta koko elinaikansa se on ollut pienessä roolissa, kuriositeettina, imperatiivisen paradigman rinnalla.

Haskell on puhdas funktionaalinen ohjelmointikieli. Puhtaus tarkoittaa sitä, että funktiot eivät sisällä lainkaan sivuvaikutuksia vaan ainoastaan palauttavat tietyn arvon. Funktiolla sanotaan olevan sivuvaikutus, mikäli se arvon tuottamisen lisäksi muuttaa ohjelman tai muistin tilaa. Tavallisin sivuvaikutus imperatiivisissa kielissä on I/O:n suoritus, eli esimerkiksi näytölle kirjoittaminen. Sivuvaikutukset ovat kätevä tapa esimerkiksi juuri I/O:n toteuttamiseen, mutta toisaalta ne myös vaikeuttavat huomattavasti automaattisen rinnakkaisuuden toteuttamista. Puhtaassa kielessä funktio palauttaa aina samoilla parametreilla saman arvon (ominaisuus jota kutsutaan viittausten läpinäkyvyydeksi), kun taas epäpuhtaassa kielessä funktion toiminta riippuu yleensä ohjelman tilasta. Juuri puhtaus on se syy, jonka takia Haskell on houkutteleva vaihtoehto automaattisen rinnakkaisuuden toteuttamiseen. Puhtaassa kielessä kaksi funktiota voidaan aina suorittaa rinnakkain mikäli niiden tulokset eivät riipu toisistaan. Tästä säieturvallisuudesta (engl. *thread safety*) johtuen kääntäjän tai ajoympäristön on helpompi toteuttaa rinnakkaisuus automaattisesti. Epäpuhtaassa kielessä asia ei ole näin yksinkertainen, vaan aliohjelmat saattavat käyttää samoja globaaleja muuttujia tai osoittimia samaan muistialueeseen mikä vaikeuttaa huomattavasti rinnakkaisuuden automaattista toteuttamista.

Haskellin ensimmäinen versio ilmestyi vuonna 1990. Haskell on ollut avoimen lähdekoodin projekti koko elinaikansa, ja tämän johdosta Haskelliin on saatavilla suuret määrät laajen-

nuksia ja ohjelmakirjastoja. Tässä tutkielmassa tutustutaan kahteen Haskell-laajennukseen: Glasgow Parallel Haskelliin ja Data Parallel Haskelliin. Molemmat lisäävät Haskelliin mahdollisuuden kirjoittaa rinnakkain suoritettavia ohjelmia. Glasgow Parallel Haskell on näistä kahdesta kypsempi ja vakaampi laajennus, jossa rinnakkaisuus toteutetaan puoli-implisiittisesti. Tämä tarkoittaa sitä, että ohjelmoija merkkaa avainsanoilla ohjelmakoodiin kohdat, joissa rinnakkaisuutta voidaan hyödyntää, mutta ajoympäristö huolehtii kaikesta rinnakkaisuuden käytännön toteutuksesta, kuten säikeiden luonnista ja laskennan jakamisesta säikeille. Data Parallel Haskell on uudempi laajennus joka keskittyy nimensä mukaisesti nimenomaan tietorinnakkaisuuden puoli-implisiittiseen toteuttamiseen. Ohjelmoija kirjoittaa ohjelman käyttäen rinnakkaisia taulukoita (esitellään luvussa 5.1) tarpeen mukaan, ja ajoympäristö luo säikeet ja jakaa näiden taulukoiden sisältämän datan ja operaatiot säikeille automaattisesti.

Tutkielman toisessa luvussa tarkastellaan yleisesti mitä tarkoitetaan rinnakkaisuudella ja miten se eroaa samanaikaisuudesta sekä tarkastellaan rinnakkaisuuden eri tyyppejä ja rinnakkaisuuteen liittyviä rajoituksia. Kolmannessa luvussa esitellään lyhyesti Haskellin historiaa ja käydään läpi Haskellin tärkeimpiä ominaisuuksia, kuten puhtaus ja laiska evaluointi. Neljännessä luvussa esitellään tarkemmin Glasgow Parallel Haskell sekä sen kääntäjä ja ajoympäristö Glasgow Haskell Compiler, joka on Haskell-kääntäjänä de facto -standardin asemassa. Viidennessä luvussa tarkastellaan Data Parallel Haskellia ja sen ominaisuuksia. Kuudennessa luvussa tutkitaan normaalin perättäissuoritukseen perustuvan Haskell-ohjelman muuntaminen rinnakkain suoritettavaksi Glasgow Parallel Haskellin avulla. Koeohjelmana on käytetty itse kirjoitettua Determinants-ohjelmaa, joka laskee matriisien determinantteja ja kirjoittaa ne tiedostoon.

2 SAMANAIKAISUUS JA RINNAKKAISUUS

Termejä samanaikaisuus (engl. *concurrency*) ja rinnakkaisuus (engl. *parallelism*) käytetään termeinä varsin paljon ristiin sekä alan kirjallisuudessa että varsinkin webin keskusteluissa. Niiden merkityserot riippuvatkin usein kirjoittajan näkökulmasta. Tutkielmassa on käytetty Simon Peyton Jonesin esittelemää määritelmää rinnakkaisuudelle ja samanaikaisuudelle [Pey02], joka on esitetty luvuissa 2.1 ja 2.2. Tässä tutkielmassa on keskitytty nimenomaan rinnakkaisuuden ja sen hyödyntämismahdollisuuksien tarkasteluun.

2.1 Samanaikaisuus

Perinteisesti tietokoneohjelma koostuu joukosta käskyjä, jotka prosessori suorittaa peräkkäin, yksi kerrallaan. Mikäli näitä käskyjä on mahdollista suorittaa rinnakkain (engl. parallel), puhutaan samanaikaisesta ohjelmasta (engl. concurrent program) [Ben06]. On huomattava, että samanaikaisuus tarkoittaa vain mahdollisuutta rinnakkaiseen suoritukseen, se ei edellytä sitä.

Rinnakkaisohjelma käyttää useita prosessoreita nopeuttamaan suoritusta. Ohjelman semantiikka on sama kuin peräkkäin suoritetun ja tulokset ovat aina samoja eli ohjelma on täysin deterministinen. Samanaikainen ohjelma taas on jo semantiikaltaan erilainen kuin peräkkäin suoritettu. Samanaikainen ohjelma sisältää useita itsenäisiä säikeitä, jotka voivat kaikki olla vuorovaikutuksessa käyttäjien kanssa ja jotka toimivat samanaikaisesti. Samanaikainen ohjelma voi toimia yhdessä prosessorissa, jolloin säikeiden samanaikaisuus on näennäistä, tai useassa jolloin suoritus on aidosti samanaikaista. Samanaikainen ohjelma on aina luonteeltaan epädeterministinen.

Joissain yhteyksissä käytetään sekaisin termejä samanaikainen ja hajautettu (engl. *distributed*). Hajautettu ohjelma sijaitsee useammalla fyysisellä koneella, jotka kommunikoivat verkon yli. Hajautettu ohjelma on usein samanaikainen, mutta ei välttämättä.

2.2 Rinnakkaisuus

Rinnakkaislaskennan johtavana ajatuksena on, että suurempi laskutehtävä jaetaan pienempiin toisistaan mahdollisimman riippumattomiin osaongelmiin, jotka voidaan laskea rinnakkain (engl. *in parallel*). Rinnakkaislaskenta on ollut käytössä supertietokoneissa jo vuosikymmeniä, mutta todellinen kiinnostus rinnakkaisuuden hyödyntämiseen tavallisissa kotikoneissa heräsi vasta kun kellotaajuuksien nostamiseen perustuva prosessorikehitys tuli, ainakin toistaiseksi, tiensä päähän kaksituhatluvun alussa, ja moniydinprosessoreista muodostui alan uusi suunta.

Vaikka rinnakkaisuus voi nopeuttaa ohjelmia huomattavasti, tuottaa se myös mittavan määrän uusia haasteita ohjelmistokehitykselle. Implisiittinen rinnakkaisuus, jossa ohjelmoija kirjoittaa ohjelman samalla tavalla kuin perinteisessä peräkkäissuorituksessa ja kääntäjä huolehtii rinnakkaisuuden toteutuksesta, on ainakin toistaiseksi jäänyt pelkäksi haaveeksi. Vastaavasti täysin eksplisiittinen rinnakkaisuus, jossa ohjelmoija on vastuussa kaikesta rinnakkaisuuden määrittelystä, on hankalaa, erittäin työlästä ja virhealtista.

2.2.1 Riippuvuudet

Ideaalitilanteessa rinnakkain suoritettavan ohjelman suoritusaika puolittuu aina kun prosessoreiden tai prosessointielementtien määrä tuplataan. Käytännössä tällainen optimaalinen tilanne on kuitenkin mahdoton, sillä kaikissa ei-triviaaleissa ohjelmissa on riippuvuuksia (engl. *dependency*), jotka rajoittavat rinnakkaisuutta. Kahden ohjelmalausekkeen välillä sanotaan olevan riippuvuus, mikäli niiden suoritusjärjestys vaikuttaa ohjelman lopputulokseen. Riippuvuudet asettavat ylärajan ohjelman suoritusnopeuden kasvattamiselle rinnakkaisuuden avulla. Riippuvuudet voidaan karkeasti jakaa kahteen eri tyyppiin: tietoriippuvuudet (engl. *data dependency*) ja kontrolliriippuvuudet (engl. *control dependency*) [Bar11]. Seuraavassa esimerkissä lausekkeen 2 tulos riippuu lausekkeen 1 tuloksesta, jolloin näiden kahden sijoituslauseen välillä on tietoriippuvuus:

```
-- Tietoriippuvuus
1:  a = 1;
2:  b = a;

-- Kontrolliriippuvuus
3:  if (c != d)
4:    c = c+d;
```

Kontrolliriippuvuudessa toisen lausekkeen tuloksesta riippuu suoritetaanko toista lauseketta lainkaan. Yllä olevassa esimerkissä lausekkeen 3 tulos määrittää suoritetaanko lauseketta 4 lainkaan, jolloin lausekkeiden 3 ja 4 välillä on kontrolliriippuvuus.

2.2.2 Amdahlin ja Gustafsonin lait

Amdahlin laki, joka on nimetty keksijänsä Gene Amdahlin mukaan, toteaa että ohjelman suurimman mahdollisen rinnakkaisuuteen perustuvan nopeutumisen määrää se osa ohjelmakoodista, joka voidaan suorittaa rinnakkain [Amd67]. Kaava (1) kuvaa Amdahlin lain.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

S kuvaa ohjelman nopeutumista suhteessa alkuperäiseen, täysin peräkkäiseen suoritukseen. P kuvaa sitä osaa koodista, joka voidaan suorittaa rinnakkain ja N kuvaa prosessointielementtien määrää. Jos mitään ei voida suorittaa rinnakkain, $P = 0$ ja $S = 1$. Jos esimerkiksi kolmasosa voidaan suorittaa rinnakkain ja prosessointielementtejä on 10, $P = \frac{1}{3}$,$N = 10$ ja $S \approx 1.43$, eli ohjelma voidaan suorittaa korkeintaan 1.43 kertaa nopeammin. Kaavasta voidaan päätellä, että rinnakkaissuorituksen skaalautuvuudella on rajansa. Tietyn rajan jälkeen prosessointielementtien määrän kasvattaminen ei enää merkittävästi lisää nopeutta, vaan ohjelman peräkkäissuoritusta vaativan osan suuruus määrää nopeutumisen ylärajan [Amd67].

Amdahlin laki näytti rajoittavan rinnakkaislaskennan mahdollisuuksia huomattavasti. Vasta pari vuosikymmentä myöhemmin, vuonna 1988, John L. Gustafson ja Edward H. Barsis esittelivät niin sanotun Gustafsonin lain. Amdahlin laki olettaa, että ratkaistavana olevan ongelman koko on muuttumaton. Gustafsonin laki esittää, että vaikka peräkkäin suoritettavat osat edelleen määräävät ohjelman absoluuttisen suoritusajan alarajan, voidaan prosessoreiden määrän ja tehon lisääntyessä samassa ajassa ratkaista aina suurempia ongelmia. Eli oletetaan vakioksi ongelman koon sijaan suoritusaika. Kaava (2) kuvaa Gustafsonin lain.

$$S = N + (1 - N) \cdot (1 - P) \tag{2}$$

Kaavassa S on ohjelman suhteellinen nopeutuminen, N on prosessoreiden määrä ja P on se osa ohjelmasta joka voidaan suorittaa rinnakkain. Jos oletetaan jälleen, että kolmasosa ohjelmasta voidaan suorittaa rinnakkain ja prosessointielementtejä on 10, on Gustafsonin lain mukaan nopeutus siis muotoa $10 + (1 - 10) \cdot (1 - \frac{1}{3}) = 4$. Suoritusnopeuden kasvu on paljon enemmän kuin Amdahlin lain ennustama 1.43 [Gus88].

2.2.3 Flynnin taksonomia

Yleisesti käytetty luokittelu prosessoriarkkitehtuureille on Michael J. Flynnin vuonna 1966 esittelemä Flynnin taksonomia. Se jakaa prosessorit kahden parametrin perusteella, tiedon ja käskyjen. Molemmilla voi olla kaksi arvoa, yksi (engl. *single*) tai monia (engl. *multiple*). Kaksi parametriä joista kummallakin voi olla kaksi arvoa muodostaa neljä prosessoriarkkitehtuurien luokkaa [Bar11] :

- **SISD, Single Instruction Single Data**
  - Perinteinen peräkkäissuoritukseen perustuva prosessori. Yksi käsky suoritetaan kerrallaan ja syötteenä toimii yksi tietovirta.
- **SIMD, Single Instruction Multiple Data**

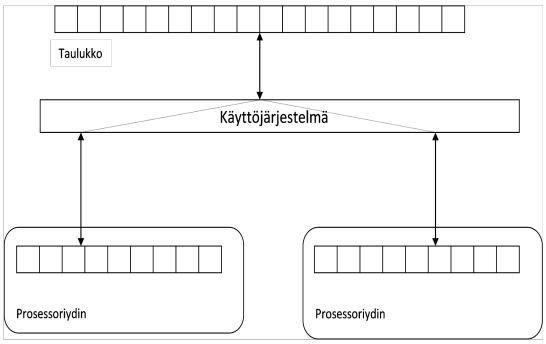
  - Yhdenlainen rinnakkaislaskentaa suorittava prosessori. Kaikki prosessointielementit suorittavat samaa käskyä, mutta eri tietovirroille. Nykyaikaiset näytönohjaimet ovat esimerkki tämäntyyppisestä prosessorista.

- **MISD, Multiple Instruction Single Data**
  - Käytännössä harvinainen prosessorityyppi, jossa useat prosessorielementit suorittavat useita käskyjä samalle tietovirralle.

- **MIMD, Multiple Instruction Multiple Data**
  - Nykyaikana yleisin prosessorityyppi. Jokainen prosessorielementti voi suorittaa eri käskyjä eri tietovirroille. Usein MIMD-arkkitehtuureissa on osana myös SIMD-tyyppinen suoritusyksikkö.

2.2.4 Tietorinnakkaisuus

Tietorinnakkaisuus (engl. *data parallelism*) on tekniikka, jossa suuri tietomäärä jaetaan pienempiin osiin, joita voidaan käsitellä rinnakkain. Käsittelyn jälkeen osat yhdistetään jälleen kokonaisuudeksi. Tietorinnakkaisuutta kutsutaan joskus myös silmukkatason rinnakkaisuudeksi (engl. *loop-level parallelism*), koska sitä esiintyy usein imperatiivisten ohjelmointikielien silmukoissa. Silmukassa käsiteltävä tietorakenne voidaan jakaa osiin (esim. yhtä moneen osaan kuin on vapaita prosessointielementtejä) ja suorittaa operaatio(t) rakenteen alkioille rinnakkain [PCL08]. Edellytyksenä tietorinnakkaisuuden hyödyntämiselle on, että suoritettavat operaatiot eivät sisällä riippuvuuksia. Tietorinnakkaisuus voi olla täysin eksplisiittistä (jossa ohjelmoija on vastuussa työn jakamisesta, säikeiden luomisesta ja kommunikoinnista), implisiittistä tai jotain näiden väliltä. Luvussa 5 esiteltävä Data Parallel Haskell on esimerkki puoli-implisiittisestä tietorinnakkaisuudesta. Ohjelmoija antaa ajoympäristölle vihjeen milloin tietorinnakkaisuutta hyödynnetään, ja ajoympäristö huolehtii rinnakkaisuuden varsinaisesta toteutuksesta. Kuva 1 havainnollistaa tietorinnakkaisuuden idean yleisellä tasolla.



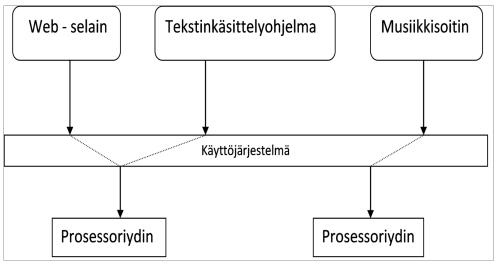
Kuva 1: Tietorinnakkaisuus

Nykyaikaiset grafiikkakortit ovat esimerkki tietorinnakkaisuutta erittäin vahvasti hyödyntävistä laitteista. Renderöinti ja pikseliefektien laskeminen ovat luonteeltaan helposti rinnakkaistettavia ongelmia ja nykyiset grafiikkakortit sisältävät satoja rinnakkaislaskentaa suorittavia laskentayksiköitä [OHL08].

2.2.5 Tehtävärinnakkaisuus

Tehtävärinnakkaisuus on kahden toisistaan riippumattoman tehtävän (jotka tosin voivat jakaa resursseja) suorittamista rinnakkain. Usein tietokoneen käyttäjällä on auki useampia ohjelmia kerrallaan, esimerkiksi web-selain sekä tekstinkäsittelyohjelma, ja nämä ohjelmat (tehtävät) suoritetaan rinnakkain. Tehtävärinnakkaisuuden erottaa moniajosta (engl. *multitasking*) nimenomaan rinnakkaisuus [Mei11]. Moniajossa perinteinen, yksiytiminen prosessori vaihtaa kontrollia tehtävien välillä niin nopeasti, että syntyy illuusio rinnakkaisuudesta. Vasta usean prosessorin tai ytimen järjestelmässä suoritus voi tapahtua todella rinnakkain. Kuva 2 havainnollistaa tehtävärinnakkaisuutta.

Kuvan 2 havainnollistama tehtävärinnakkaisuus on sovellustason (engl. *application-level*) rinnakkaisuutta, jossa käyttöjärjestelmä huolehtii tehtävien jakamisesta prosessoreiden kes-



Kuva 2: Tehtävärinnakkaisuus

ken. Itse sovellusten ei tarvitse tietää rinnakkaisuudesta mitään. Jos yksittäisen sovelluksen tehtäviä (esim. verkkoliikenne, käyttöliittymätoiminnot, lokitietojen tallennus) halutaan suorittaa rinnakkain, tehtävät pitää olla eksplisiittisesti jaettuna suoritussäikeisiin (tai puoli-implisiittisesti kuten luvussa 4 esiteltävässä Glasgow Parallel Haskellissa). Ohjelmoijan pitää myös huolehtia säikeiden välisestä kommunikaatiosta ja synkronoinnista esimerkiksi lukkojen avulla [Mei11].

2.2.6 Säikeet

Käyttöjärjestelmä suorittaa erilliset ohjelmat omina prosesseinaan. Prosesseilla on kaikilla oma tilansa, oma muistialueensa ja omat osoiteavaruutensa. Jokainen prosessi sisältää vähintään yhden säikeen (engl. *thread*). Säie kuvaa ohjelman suorituspolun prosessin sisällä. Prosessit voivat sisältää useita säikeitä, jotka voivat suorittaa tehtäviä samanaikaisesti tai rinnakkain prosessin sisällä. Säikeet eivät sisällä tilatietoa vaan jakavat isäntäprosessinsa tilan, muistialueen, osoiteavaruuden sekä muut resurssit [Lee06].

Nykyaikaiset prosessorit tukevat laitetasolla monisäikeisyyttä (engl. *multithreading*), ja monisäikeiset ohjelmat ovatkin saavuttaneet de facto -standardin aseman samanaikaisten ja rinnakkaisten ohjelmien toteutuksessa vaikka säikeitä ohjelmoijan työkaluina onkin kritisoitu jonkin verran. Kriitikoiden mielestä ihmisen on lähes mahdotonta ymmärtää kunnolla isompien monisäikeisten ohjelmien toimintaa. Heidän mukaansa on olemassa helpommin

ymmärrettäviä ja käyttäjäystävällisempiä tapoja toteuttaa rinnakkaisuutta tavallisissa ohjelmissa ja säikeitä tulisi käyttää ainoastaan laitetasolla laskentakuorman jakamiseen prosessoreille [Lee06]. Myös Haskell käyttää säikeitä rinnakkaisuuden käytännön toteutukseen, joskin Haskell-ajoympäristö tekee suurimman osan säikeiden luontiin ja synkronointiin liittyvästä työstä ohjelmoijan puolesta.

3 HASKELL

3.1 Haskellin kehitys

Funktionaalisen ohjelmoinnin juuret sijoittuvat Alonzo Churchin 1930-luvulla kehittämään lambda-kalkyyliin. Lambda-kalkyyli loi formaalin tavan kuvata funktioiden laskettavuutta (engl. *computability*) ja oli lisäksi säännöiltään suhteellisen pieni ja yksinkertainen [Roj97]. Vaikka lambda-kalkyyliä voidaan jälkikäteen tarkasteltuna pitää ensimmäisenä funktionaalisena ohjelmointikielenä, funktionaalisten ohjelmointikielien historian mainitaan kuitenkin usein alkaneen vuonna 1958 julkaistusta Lisp-ohjelmointikielestä. Lispin jälkeen esiteltiin myös muita funktionaalisia kieliä [Hud89], mutta 1970- ja 1980-lukujen vaihteessa esitelty ns. laiska evaluointi (engl. *lazy evaluation*) herätti kiinnostuksen funktionaaliseen ohjelmointiin toden teolla, ja 1980-luvun aikana uusia funktionaalisia ohjelmointikieliä ilmestyi tiheään tahtiin [HHP07].

Syyskuussa 1987 funktionaalisten ohjelmointikielten ja tietokonearkkitehtuurien konferenssin (engl. *Conference on Functional Programming Languages and Computer Architecture*) yhteydessä alan johtavat tutkijat pitivät kokouksen funktionaalisten ohjelmointikielten senhetkisestä tilasta. Kielten kehitys oli jakautunut pahasti ja hyvin samankaltaisia, puhtaita ja laiskoja funktionaalisia kieliä oli useita, jopa kymmeniä. Tutkijat olivat yhtä mieltä siitä, että tämä kielten paljous haittaa funktionaalisen ohjelmoinnin yleistymistä ja päättivät uuden komitean perustamisesta, jonka tehtävänä olisi uuden, yhteisen kielen kehittäminen.

Aluksi komitean tarkoituksena oli valita jokin valmis kieli ja kehittää sitä edelleen. David Turnerin kehittämä Miranda oli heidän mielestään paras vaihtoehto. Turner ei kuitenkaan hyväksynyt Mirandan käyttöä tähän tarkoitukseen, joten komitea päätti kehittää kokonaan uuden kielen. Kehittäjien mukaan tämä osoittautui lopulta hyväksi ratkaisuksi, koska he saattoivat kehitellä radikaaleja lähestymistapoja ja kokonaan uusia ratkaisuja, kuten tyyppiluokat. Komitean ensimmäisessä virallisessa tapaamisessa uuden kielen nimeksi sovittiin

Haskell, amerikkalaisen matemaatikko Haskell Curryn mukaan, jonka työ kombinatorisen logiikan parissa on vaikuttanut vahvasti myös funktionaalisen ohjelmoinnin kehitykseen [HHP07].

Haskellin ensimmäinen versio (engl. *The Haskell Report version 1.0*) ilmestyi huhtikuun ensimmäisenä päivänä 1990. Ensimmäisen version ilmestymisen ja sitä edeltäneiden useiden kokousten jälkeen Haskellin kehitystyö jatkui tiiviinä eri tahoilla ja komitean jäsenet (ja useat muut, sillä komitean postituslista oli julkinen) pitivät yhteyttä ja kävivät keskustelua lähinnä sähköpostin välityksellä. Haskellin websivusto (http://www.haskell.org) julkaistiin vuonna 1994. Helmikuussa 1999 julkaistiin Haskell 98 [Pey12], joka oli ensimmäinen kerta kun kielen tietty versio nimettiin. Nimeäminen oli samalla epävirallinen standardointi ja helpotti kielen tukemista ilman pelkoa jatkuvista muutoksista. Tämän epävirallisen standardoinnin myötä päätettiin myös Haskell-komitean lopettamisesta, jotta kielen kehittyminen voisi jatkua useisiin eri suuntiin.

Kirjoitushetkellä Haskellin uusin versio on Haskell 2010, joka ilmestyi loppuvuonna 2009. Tällä hetkellä Haskellin kehitys tapahtuu vapaaehtoisen yhteisön voimin, johon kuka tahansa voi osallistua. Yhteisön tavoitteena on julkaista kerran vuodessa uusi versio ja jokaista versiota varten yhteisöstä muodostetaan komitea, joka seuraa kehitystyötä ja lopulta päättää mitä ominaisuuksia uuteen versioon otetaan mukaan. Haskell 2010 oli ensimmäinen versio, joka julkaistiin tällä menettelyllä [Has12].

3.2 Puhtaus ja laiska evaluointi

Ohjelmointikielen suoritusstrategia (engl. *evaluation strategy*) on joukko sääntöjä, joiden mukaan lausekkeita suoritetaan. Suoritusstrategia määrää erityisesti funktioiden ja operaattorien käyttäytymisen eli missä järjestyksessä ja milloin operaation osapuolet tai funktion parametrit evaluoidaan, sekä funktioiden tapauksessa milloin parametrit sijoitetaan itse funktioon.

Lähes kaikki imperatiiviset ohjelmointikielet, esimerkiksi C/C++ ja Java, käyttävät suori-

tusstrategiana tiukkaa evaluointia (engl. *strict evaluation*). Tiukassa evaluoinnissa lausekkeet evaluoidaan heti, kun ne on kiinnitetty johonkin muuttujaan ja funktion parametrit evaluoidaan aina ennen funktion suoritusta. Ohjelmoija voi siis koodin jäsentelyllä vaikuttaa lausekkeiden suoritusjärjestykseen. Toisaalta tiukka evaluointi saattaa aiheuttaa turhaa laskentaa, kun evaluoidaan parametreja tai lausekkeita, joita ei koskaan käytetä. Koodin optimointi tältä osin jää siis ohjelmoijan harteille.

Tiukkaan evaluointiin kuuluu useita strategioita funktion parametrien evaluoimiseen. Näistä käytetyin on ns. call-by-value (tai pass-by-value), jossa funktion parametrit evaluoidaan ja niiden arvot kopioidaan muuttujiin funktiossa. Näin ollen funktiot eivät muuta alkuperäisiä parametreja, vaan ainoastaan käyttävät kopioita niiden arvoista. Toinen yleisesti käytetty tiukka strategia on call-by-reference, jossa funktio voi muuttaa myös alkuperäistä parametria.

Laiska evaluointi (engl. *lazy evaluation*) poikkeaa tiukasta siten, että lauseke evaluoidaan vasta sitten, kun jokin toinen funktio kutsuu sitä. Parametreja ja lausekkeita evaluoidaan lisäksi vain sen verran, kuin kutsuva funktio tarvitsee suorittaakseen oman laskentansa. John Hughes käyttää seuraavaa esimerkkiä laiskan evaluoinnin havainnollistamiseen artikkelissaan funktionaalisen ohjelmoinnin hyödyistä [Hug89]:

Mikäli f ja g ovat funktiota, on $g(f(\text{syöte}))$ ohjelma, jossa funktio f laskee tuloksensa ja sitä käytetään syötteenä funktiolle g. Perinteisissä ohjelmointikielissä f evaluoitaisiin ensin kokonaan ja tulos tallennettaisiin ennen kuin se annettaisiin syötteenä funktiolle g. Tämä voi viedä paljon tilaa ja aikaa. Haskellissa suoritusjärjestys ei kuitenkaan ole tällainen, vaan funktiota f ryhdytään suorittamaan vasta kun g yrittää lukea syötettään. Lisäksi funktiota f evaluoidaan vain niin kauan, kunnes g on saanut tarvitsemansa syötteen ja funktion f suoritus keskeytetään siihen asti kunnes g yrittää lukea toista syötettä. Mikäli funktion g suoritus päättyy ilman että se on lukenut kaikkia funktion f tuloksia, funktion f suoritus keskeytetään kokonaan. Funktio f voi olla jopa päättymätön, sillä sen suoritus keskeytetään joka tapauksessa kun funktion g suoritus päättyy. Siis silmukan runko ja lopetusehto voidaan eriyttää toisistaan. Tämä suoritustapa suorittaa funktiota f vain silloin ja sen verran kuin sitä välttämättä tarvitaan ja siksi sitä sanotaan laiskaksi evaluoinninksi [Hug89].

Laiskan evaluoinnin ilmeinen hyöty on turhan laskennan välttäminen. Toinen todella tärkeä hyöty on vahva modularisaatio, jonka edellä mainittu silmukan rungon ja lopetusehdon eriyttäminen mahdollistaa. Hughes demonstroi tätä modularisaatiota artikkelissaan muutamin esimerkein.

Laiskaan evaluointiin kuuluu ns. call-by-name-strategia, jossa funktion parametreja ei evaluoida erikseen, vaan ne sijoitetaan suoraan funktioon lambda-kalkyylin osoittamalla tavalla. Tästä seuraa, että mikäli parametria ei tarvita funktion suorittamisessa, sitä ei koskaan evaluoida. Jos parametria tarvitaan useassa kohtaa, se evaluoidaan erikseen joka kerta. Yleisempi strategia laiskan evaluoinnin kielissä, joita myös Haskell edustaa, on ns. call-by-need-strategia, joka toimii muuten samoin kuin call-by-name, mutta se muistaa jo aiemmin evaluoidut arvot (engl. *sharing*). Jos parametri evaluoidaan kerran, sen arvo tallennetaan ja sitä voidaan käyttää myöhemmin uudelleen. Mitään termiä ei siis evaluoida kuin korkeintaan kerran [Hud89].

Haskellin käyttämä laiska, call-by-need suoritustrategia on lähes aina hitaampi kuin tiukan evaluoinnin call-by-value. Tämä johtuu ylimääräisestä kirjanpidosta, jonka evaluoinnin viivyttäminen siihen asti kunnes termiä tarvitaan ja termin evaluoinnin jälkeen kaikkien sen esiintymien korvaaminen tuloksella (jotta mitään termiä ei evaluoida kuin korkeintaan kerran) aiheuttaa [HHP07].

Laiskan suorituksen suurin kritiikki ei aiheudu kuitenkaan tästä hitaudesta verrattuna perinteisiin kieliin kuten C, koska tämä kirjanpitotaakka on aina vakio. Suurimman kritiikin kohteena on ohjelmien vaikeasti ennustettava tilankäyttö, joka ei puolestaan ole vakio, vaan voi vaihdella rajustikin. Tilankäytön huonon ennustettavuuden vuoksi Haskelliin on lisätty joitakin tiukan evaluoinnin ominaisuuksia, kuten esimerkiksi luvussa 4 esiteltävä operaattori ’pseq’ [HHP07].

Funktiolla sanotaan olevan sivuvaikutus tai sivuvaikutuksia, jos se tuloksen tuottamisen lisäksi muuttaa ohjelman tilaa. Tilan muuttamista on esimerkiksi globaalin muuttujan muokkaaminen, omien parametrien muokkaaminen, I/O-laitteelle kirjoittaminen tai siltä lukeminen, poikkeuksen nostaminen tai jonkin sivuvaikutuksia sisältävän funktion kutsuminen.

Perinteisten ohjelmointikielten kuten C/C++ ja Java (ja kaikkien muidenkin imperatiivisten kielten) toiminta perustuu nimenomaan näihin ohjelman tilan muutoksiin eli sivuvaikutuksiin. Sivuvaikutukset ovatkin ylivoimaisesti käytetyin tapa toteuttaa I/O eli ohjelman vuorovaikutus käyttäjän kanssa. Sivuvaikutusten takia funktioiden tulos ja toiminta riippuu siitä milloin ne suoritetaan, eli tulos riippuu suoritusjärjestyksestä.

Laiskan evaluoinnin seurauksena ohjelmoija ei voi tietää milloin joku tietty funktio tai lauseke suoritetaan vai suoritetaanko sitä ollenkaan. Tämän seurauksena funktion sivuvaikutusten käyttäminen esimerkiksi I/O:n tuottamiseen, ainakaan luotettavasti, on käytännössä mahdotonta. Näin ollen Haskell ei sisällä lainkaan sivuvaikutuksia, eli se on puhdas kieli (engl. *pure language*). Funktiokutsu Haskellissa samoilla parametreilla palauttaa aina täsmälleen saman arvon, riippumatta siitä missä vaiheessa ohjelman suoritusta sitä kutsutaan. Tätä ominaisuutta kutsutaan viittauksen läpinäkyvyydeksi (engl. *referential transparency*) [HHP07].

Puhtauden tärkeimpiä etuja ovat ohjelmien huomattava selkeytyminen, vaikeasti havaittavien muuttujavirheiden eliminointi, algoritmien (mahdollinen) yksinkertaistaminen ja koodin automaattisen optimoinnin helpottaminen. Tämän tutkielman kannalta viimeisenä mainittu on tärkeä, sillä juuri puhtaus mahdollistaa ohjelmien automaattisen rinnakkaistamisen kääntäjän toimesta.

Haskellissa, kuten muissakin puhtaissa kielissä, I/O-toimintojen toteuttaminen aiheutti kehittäjille päänvaivaa. Käytännössä I/O:n toteuttaminen vaatii sivuvaikutuksia ja tilan muutoksia, mutta miten toteuttaa ne puhtaassa kielessä? Lopulta ongelma ratkaistiin monadien avulla, joilla voitiin toteuttaa tilamuutokset pitämällä kieli silti puhtaana [PJW93]. Haskellin kehittäjät pitävät nimenomaan monadista I/O:ta Haskellin tärkeimpänä kontribuutiona funktionaaliselle ohjelmoinnille. Laiskan evaluoinnin ansiosta kielestä tuli puhdas ja ennen kaikkea se pysyi puhtaana. Halu säilyttää puhtaus taas johti monadisen I/O:n kehittämiseen [HHP07].

3.3 Funktiot

Seuraavissa luvuissa esitettävät koodiesimerkit on tehty interaktiivisessa GHCi Haskell-tulkissa. Käyttöjärjestelmänä oli Windows 7. GHCi tulkitsee käyttäjän antaman Haskell-lausekkeen ja näyttää tuloksen. Ohjelma toimii siis hyvin samaan tapaan kuin perinteinen laskin. Mikäli käyttäjä haluaa evaluoida useampia lausekkeita tai kirjoittaa omia funktioitaan, ne on kirjoitettava erillisellä ohjelmalla ja ladattava GHCi-ympäristöön :load-komennolla. GHCi soveltuu vain lausekkeiden tulkitsemiseen, sillä ei voi kääntää ohjelmia ajettavaan muotoon.

Funktio on Haskellin, kuten kaikkien funktionaalisten kielten, keskeisin käsite. Haskell-ohjelma koostuu joukosta funktiomääritelmiä. Funktion määritelmä koostuu kahdesta osasta, funktion tyypin määritelmästä (engl. *type signature*) ja varsinaisen laskennan määritelmästä. Tyypin määritelmä määrittää parametrien ja tuloksen tyypin. Tyypin määritelmä on valinnainen osa. Mikäli ohjelmoija jättää sen kirjoittamatta, kääntäjä päättelee sen itse funktiossa käytetyistä operaatioista. Seuraava funktio määrittelee 'nelio'-nimisen funktion, joka saa parametrinaan yhden kokonaisluvun ja tuottaa tuloksenaan yhden kokonaisluvun. Funktio kertoo parametrina saamansa luvun itsellään ja palauttaa tuloksen. Kommentit alkavat "- -" - merkillä:

```
-- Funktion tyypin määrittely
nelio :: Int → Int


-- Laskennan määrittely
nelio x = x*x
```

GHCi-ympäristössä tämän funktion käyttäminen onnistuu lataamalla funktion sisältämä tekstitiedosto käytettäväksi, jonka jälkeen sen käyttäminen onnistuu funktion nimellä. Tässä tapauksessa tiedoston nimi on 'nelio.hs':

```
Prelude> :load nelio.hs
Main> nelio 3
9
```

3.4 Listat

Listat ovat funktioiden ohella Haskell-ohjelmoijan tärkeimpiä työkaluja. Aivan kuten imperatiivisten kielten taulukot, ne ovat keino käsitellä useampia arvoja kerrallaan. Haskellin listat muistuttavat syntaksiltaan taulukoita ja voivat sisältää melkein mitä tahansa arvoja, mutta yksi lista voi sisältää vain yhden tyyppisiä arvoja. Aivan kuten taulukoissakin, hakasulkeet määrittelevät listan ja arvot erotetaan toisistaan pilkuilla. Tässä ensimmäinen lista sisältää kokonaislukuja ja toinen merkkijonoja:

```
Prelude> [1,2,3,4]
[1,2,3,4]

Prelude> ["Tama","on","lista"]
["Tama","on","lista"]
```

Ylläolevat listat on määritelty koko lista kerrallaan. Toinen tapa rakentaa listoja on Haskellin cons-operaattori, eli kaksoispiste (:). Cons-operaattorilla lisätään arvoja tyhjään tai jo olemassa olevaan listaan:

```
Prelude> 0:[1,2,3]
[0,1,2,3]

Prelude> 0:1:2:3:[]
[0,1,2,3]
```

Cons-operaattorilla ei voi liittää kahta listaa toisiinsa, vaan ainoastaan lisätä yhden alkion listan alkuun. Listat voivat tosin olla moniulotteisia, eli lista voi sisältää listoja. Kaksiulotteisen listan tapauksessa yksi alkio on siis tavallinen lista:

```
Prelude> [1,2]:[[3,4],[5,6]]
[[1,2],[3,4],[5,6]]
```

Haskell sisältää listojen muodostamiseen myös tehokkaan ja ilmaisuvoimaisen listakonstruktorin (engl. *list comprehension*):

```
Prelude> [n | n←[1,2,3,4], n>2]
[3,4]
```

Tämän syntaksi on jokseenkin tuttu matematiikasta. Se luetaan siten, että n saa vuorollaan kaikki arvot taulukosta $[1, 2, 3, 4]$. Jokaisen arvon kohdalla testataan pilkun jälkeen tuleva ehto tai ehdot, tässä tapauksessa $n > 2$. Mikäli arvo täyttää ehdon, se lisätään varsinaiseen tuloslistaan siinä muodossa joka on määritelty ennen pystyviivaa, tässä tapauksessa sellaisenaan arvona n.

3.5 Korkeamman tason funktiot

Haskellissa funktiot voivat saada parametreina toisia funktiota ja myös palauttaa toisia funktioita. Tällöin puhutaan korkeamman tason funktioista. Esimerkiksi seuraava funktio saa parametrina funktion ja suorittaa tämän funktion kahdesti toisena parametrina saamalleen arvolle.

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

Sulkeet funktion tyypin määrittelyssä ovat tärkeät, koska ne ilmaisevat että ensimmäinen parametri on funktio, joka ottaa jonkin tyyppisen arvon ja palauttaa samaa tyyppiä olevan arvon. Korkeamman tason funktioita käytetään kuten tavallisiakin funktiota, eli funktion nimen jälkeen listataan parametrit välilyönnein erotettuna.

```
-- Apufunktio, joka kertoo parametrina saamansa arvon kahdella
multip :: Num a ⇒ a → a
multip x = x*2

*Main> twice multip 6
24
```

3.6 Hahmonsovitus

Hahmonsovituksessa (engl. *pattern matching*) määritellään erilaisia muotoja tai kaavoja, joita funktion parametrit noudattavat. Eri hahmoille voidaan kirjoittaa erilainen määritelmä samasta funktiosta ja aina käytetään sitä funktion määritelmää, joka sopii saatuun parametriin.

```
-- Funktio, joka laskee listan alkioiden määrän
listLength :: [a] → Int
listLength [] = 0
listLength (x:xs) = 1 + listLength xs
```

Tässä esimerkissä on käytetty hahmonsovitusta listoille. Funktio palauttaa eri arvon riippuen siitä minkälaisen listan se saa parametrina. Kun funktiota kutsutaan niin määritellyt hahmot käydään läpi yksitellen ylhäältä alas. Esimerkin tapauksessa ensin tarkistetaan onko parametrina tyhjä lista ja sen jälkeen onko listassa yksi tai useampia alkioita. Hahmojen tarkistus päättyy kun kohdataan ensimmäinen sopiva hahmo ja tätä hahmoa vastaava funktio suoritetaan. Jos yhtään sopivaa hahmoa ei löydy, funktion suoritus kaatuu virheeseen. Esimerkissä on käytetty listoja, mutta hahmonsovitusta voidaan käyttää lähes minkä tahansa tyypin kanssa [Lip11]. Hahmonsovitus aiheuttaa oletuksena funktion parametrin evaluoinnin vähintään heikkoon normaalimuotoon (ks. luku 4.2) [Pey12].

3.7 Lambdat

Nimettyjen funktioiden lisäksi Haskellissa on myös mahdollista käyttää nimettömiä funktioita eli lambda-funktioita. Tällaiset funktiot ovat hyödyllisiä kun funktiota tarvitaan vain kerran, yleensä parametrina korkeamman tason funktiolle. Seuraavassa esimerkkissä kappaleessa 3.5 käytetty esimerkki korkeamman tason funktiosta on esitetty lambda-funktion avulla, jolloin erillistä apufunktiota ei tarvita:

```
*Main> twice (\x → x*2) 6
24
```

Lambda -funktion parametrit esitellään '\'-merkin jälkeen välilyönnein eroteltuna aivan kuten tavallisen funktion parametrit. Parametrit erotetaan funktion varsinaisesta määrittelystä '->'-symbolilla. Yleensä lambda-funktiot ympäröidään suluilla koodin luettavuuden parantamiseksi. Lambda-funktioissa on mahdollista käyttää hahmonsovitusta, mutta yhdelle parametrille ei voi määritellä kuin yhden hahmon, toisin kuin tavallisissa funktioissa [Lip11].

3.8 Omat tietotyypit

3.8.1 Tyyppisynonyymit

Haskellissa käyttäjä voi luoda omia tyyppejä useammalla eri tavalla. Yksi tapa on luoda synonyymi jollekin jo olemassaolevalle tyypille käyttämällä avainsanaa 'type'.

```
type Name = String
type D3Vector = (Double, Double, Double)
```

Tyyppien synonyymit eivät määritä uutta tyyppiä, vaan määrittävät vaihtoehtoisen nimen tietotyypille. Synonyymien tarkoitus on tehdä koodista selkeämpää ja helpommin ymmärrettävämpää ihmiselle ja joissakin tapauksissa vähentää ohjelmoijan kirjoitustyötä, jos alkuperäinen tyyppi on työläs kirjoittaa.

3.8.2 Algebralliset tietotyypit

Kokonaan uusi tietotyyppi määritellään avainsanalla 'data'. Bool-tyyppi on määritetty standardikirjastossa seuraavalla tavalla [OGS08]:

```
data Bool = False | True
```

Avainsanan ’data’ jälkeen esitellään tyyppikonstruktori (engl. *type constructor*) eli uuden tyypin nimi, joka tässä tapauksessa on Bool, ja ’=’-merkin jälkeen luetellaan tyypin arvokonstruktorit (engl. *value constructors*), jotka erotellaan toisistaan loogisella tai-operaattorilla ’|’. Eli Bool voi saada joko arvon False tai True. Tällaista tyyppiä, jolla on useampi arvokonstruktori sanotaan algebralliseksi tietotyypiksi (engl. *algebraic datatype*). Arvokonstruktorit ja tyyppikonstruktorit voivat molemmat saada nolla (kuten tyypin Bool tapauksessa) tai useampia parametreja.

```
-- Arvokonstruktori saa kolme Int -tyyppistä parametria
-- kuvaamaan päivää, kuukautta ja vuotta.
data MyDate =  MyDate Int Int Int

-- Tyyppikonstruktori saa yhden tyyppiparametrin.
data Maybe a = Nothing | Just a
```

Standardikirjastossa on määritelty tyyppi ’Maybe a’, jota käytetään usein operaatioissa jotka voivat epäonnistua. Jos arvo- tai tyyppikonstruktorille on tyypin määrittelyssä annettu parametreja, on ne kaikki täsmennettävä tyyppiä käytettäessä. Esimerkiksi ’Maybe’ ei yksinään ole mikään tyyppi, vaan se vaatii aina toisen tyypin parametrikseen, kuten esim. ’Maybe Int’ tai ’Maybe String’, jotka ovat molemmat omia tyyppejään. Tyypistä, jolla on tyyppiparametreja käytetään joskus myös nimeä konteksti (engl. *context*) [Lip11]. Sanotaan, että ’a’:n konteksti on ’Maybe’. Käyttäjän määrittelemiä algebrallisia tietotyyppejä voidaan käyttää funktioissa aivan kuten Haskellin omia tietotyyppejä ja niihin voidaan tehdä hahmonsovitusta [OGS08].

```
-- Haetaan kuukausi päivämäärästä
monthFromMyDate :: MyDate → Int
monthFromMyDate (MyDate _ x _) = x

-- Haetaan String -arvo Maybe String -tyypistä
fromMaybe :: Maybe String → String
fromMaybe (Just xs) = xs
fromMaybe Nothing = ""
```

3.9 Tyyppiluokat

Haskellin tyyppiluokat määrittävät eri tyypeille yhteistä käyttäytymistä. Idealtaan ne ovat samankaltaisia kuin kuin Javan tai C#:n rajapinnat (engl. *interface*). Tyyppiluokka määrittelee joukon funktioita, jotka luokkaan kuuluvien tyyppien täytyy toteuttaa. Luokan määrittelyssä voidaan halutuille funktioille määritellä myös oletustoteutus. Ohjelmoija voi halutessaan kuitenkin kirjoittaa oman toteutuksensa myös funktioille, joilla on oletustoteutus. Ohjelmoijan oma toteutus ylikirjoittaa aina oletustoteutuksen. Yksi Haskellin yleisimmistä tyyppiluokista on Eq, joka määrittää yhtä- ja erisuuruuden tyypeille. Eq on määritelty standardikirjastossa seuraavasti:

```
class Eq a where
 (==) :: a → a → Bool
 (/=) :: a → a → Bool
 x == y = not (x /= y)
 x /= y = not (x == y)
```

Tässä määritelmässä 'a' on tyyppiparametri. Eq-tyyppiluokka määrittelee kaksi funktiota, yhtäsuuruusvertailun (==) ja erisuuruusvertailun (/=). Molemmille funktioille on määritetty oletustoteutus, mutta tämä ei ole pakollista. Ainoastaan funktioiden tyypinmäärittelyt ovat pakollisia. Oletustoteutuksena yhtäsuuruus on erisuuruuden negaatio ja päinvastoin, jolloin ohjelmoijan täytyy toteuttaa vain toinen funktioista tehdäkseen omasta tyypistään Eq-luokan jäsenen.

```
-- Määritellään oma tietotyyppi
data Shapes = Circle | Square | Triangle

-- Tehdään tästä tyypistä Eq -luokan jäsen
instance Eq Shapes where
Circle   == Cirle    = True
Square   == Square   = True
Triangle == Triangle = True
       _ == _        = False
```

Omasta tyypistä tehdään tyyppiluokan jäsen avainsanalla ’instance’ ja korvaamalla luokkamäärittelyn tyyppiparametri varsinaisella tyypillä. Tämän jälkeen määritellään kaikkien luokkaan kuuluvien funktioiden käyttäytyminen tällä tyypillä. Tyyppiluokista voi tehdä myös aliluokkia lisäämällä luokan määrittelyyn luokkarajoitteen (engl. *class constraint*) [Lip11].

3.10 Monadit

Monadit ovat yleisiä ja monikäyttöisiä tyyppejä Haskellissa. Alun perin ne kehitettiin keinoksi suorittaa I/O:ta Haskellin laiskassa ja puhtaassa ympäristössä, kuten luvussa 3.2 mainittiin. Monadeja käytetään edelleen eniten suoritusjärjestyksen määrittämiseen ja I/O:n suorittamiseen. Monadien yleiskäyttöisen luonteen takia niillä on kuitenkin paljon muitakin käyttökohteita. Yleisellä tasolla monadien avulla tyyppiin ’m a’, missä ’m’ on tyypin nimi ja ’a’ sen tyyppiparametri (eli ’a’:n konteksti on ’m’, kuten luvussa 3.8.2 mainittiin), voidaan käyttää funktiota, joka ottaa kontekstittoman parametrin ’a’ ja palauttaa jonkin arvon ’b’ samassa kontekstissa ’m’. Monad-tyyppiluokka on määritelty standardikirjastossa seuraavasti:

```
class Monad m where

 return :: a → m a

 (»=)  :: m a → (a → m b) → m b

 (»)   :: m a → m b → m b
 a » f  = a »= \_ → f

 fail :: String → m a
 fail msg = error msg
```

Funktio ’return’ ei tarkoita Haskellissa samaa kuin useimmissa muissa ohjelmointikielissä. Se ei keskeytä tai päätä funktion suoritusta, vaan ainoastaan käärii parametrina saamansa arvon tiettyyn kontekstiin. Tällaista arvon uuteen kontekstiin käärimistä kutsutaan usein nostamiseksi (engl. *lifting*). Seuraavana on funktio (»=), joka ottaa parametreina monadisen arvon ja funktion, jonka parametri on tavallinen kontekstiton arvo, mutta joka palauttaa monadisen arvon. Funktio (») jättää toisen parametrinsa huomiotta ja palauttaa vain toisena parametrina saamansa monadisen arvon. Tälle funktiolle on määritelty oletustoteutus ja sitä harvoin ylikirjoitetaan uusia monadeja luotaessa. Funktiota ’fail’ ei yleensä käytetä koodissa, vaan Haskell käyttää sitä do-notaatiossa (ks. luku 3.10.1) virheestä ilmoittamiseen. Monadit mahdollistavat toimintojen ketjuttamisen ja funktioiden suoritusjärjestyksen määräämisen (»=)-funktion avulla, jonka takia monadeja käytetään I/O:n toteuttamiseen [Lip11]. Yksi käytetyimmistä monadeista on todennäköisesti luvussa 3.8.2 esitelty ’Maybe’. Seuraavassa esimerkissä on esitelty miten ’Maybe’ toteuttaa Monad-tyyppiluokan vaatimat funktiot. Samassa esimerkissä on esitelty ’maybeAdd’-funktio, joka laskee yhteen nollasta poikkeavia kokonaislukuja. Tämän funktion avulla demonstroidaan toimintojen ketjutusta (»=)-funktion avulla. Toimintojen ketjutus tehdään yleensä käytännössä do-notaatiota käyttäen, joka esitellään seuraavassa luvussa.

```
-- Maybe -tyyppi on monadi
instance Monad Maybe where

return x  = Just x

Nothing >>= f = Nothing
Just x >>= f  = f x
fail _        = Nothing

-- Funktio laskee kaksi kokonaislukua yhteen, paitsi jos toinen on nolla
maybeAdd :: Int → Int → Maybe Int
maybeAdd _ 0    = Nothing
maybeAdd 0 _    = Nothing
maybeAdd n1 n2  = Just (n1+n2)
```

```
-- Toimintojen ketjutus (»=) funktion avulla
*Main> maybeAdd 1 2 »= maybeAdd 2 »= maybeAdd 3
Just 8

*Main> maybeAdd 1 2 »= maybeAdd 2 »= maybeAdd 0
Nothing
```

3.10.1 Do-notaatio

Monadeille on määritelty oma erikoissyntaksinsa, jota kutsutaan do-notaatioksi avainsanan 'do' johdosta. Do-notaatio muistuttaa imperatiivisia ohjelmointikieliä, joissa toiminnot suoritetaan siinä järjestyksessä kuin ne kirjoitetaan. Imperatiiviselta näyttävästä rakenteesta huolimatta do-notaatio on kuitenkin vain monadien ketjutusta (»=), (»), 'return'-funktioita ja lambda-abstraktiota käyttäen. Do-notaatiossa jokaisen rivin on oltava monadinen arvo tai arvon nimeäminen [Lip11]. Nimettävä arvo voi olla joko let-lohko, jossa nimetään yksi tai useampia ei-monadisia arvoja tai monadisen arvon nimeäminen. Monadisen arvon nimeäminen tapahtuu (<-)-operaattorilla, jonka käyttö ilmenee seuraavasta esimerkistä.

```
-- Edellisessä luvussa esitetyn maybeAdd-funktion ketjutus
-- do-notaatiota käyttäen.
testDo :: Maybe Int
testDo = do
          x ← maybeAdd 1 2
          y ← maybeAdd 3 x
          return y
```

Do-notaatio on huomattavasti yleisemmin käytetty tapa esittää toimintojen ketjutus kuin (»=)-funktion suora käyttö, koska do-notaatio on selkeämpi hahmottaa ja intuitiivisempi käyttää.

4 GLASGOW PARALLEL HASKELL

Glasgow Parallel Haskell on perinteisen Haskellin lisäosa, jolla rinnakkaisuutta voidaan toteuttaa puoli-eksplisiittisesti: ohjelmoija antaa ajoympäristölle vihjeitä mahdollisista rinnakkain evaluoitavista osista ohjelmassa, mutta ajoympäristö huolehtii kaikesta rinnakkaisuuden vaatimasta käytännön koordinaatiosta kuten säikeiden luonnista ja tuhoamisesta tai työn jakamisesta eri säikeille.

4.1 GHC-ajoympäristö

Glasgow Haskell Compiler (GHC) on laajin ja eniten käytetty Haskell-kääntäjä. Se tukee myös Glasgow Parallel Haskellia (GpH). Ennen GpH:n tarkempaa tutkimista tarkastellaan ensin, kuinka ohjelmien rinnakkaissuoritus pääpiirteissään tapahtuu Glasgow Haskell Compilerissa.

Ohjelman suorituksen aikana GHC-ajoympäristö voi luoda jopa miljoonia keveitä Haskell-säikeitä, jotka multipleksataan muutaman käyttöjärjestelmäsäikeen suoritettavaksi. Jokaisen Haskell-säikeen tila ja sen kutsupino (engl. *call stack*) säilytetään ns. tilaoliossa (engl. *Thread State Object*), joka sijaitsee kekomuistissa (engl. *heap*). Glasgow Parallel Haskellissa ohjelmoija käyttää tiettyjä operaattoreita merkitsemään ne lausekkeet, jotka voidaan suorittaa rinnakkain. Tästä käytetään alan kirjallisuudessa nimitystä lausekkeen kipinöinti (engl. *sparking*). Mikäli tämän kipinöidyn, evaluoimattoman lausekkeen (eli tyngän, engl. *thunk*), evaluointiin on prosessori vapaana, niin ajoympäristö muodostaa tätä varten uuden Haskell-säikeen. Varsinaisia työtä suorittavia käyttöjärjestelmäsäikeitä on suunnilleen saman verran kuin prosessoreita, mutta jokaista prosessoria kohti on täsmälleen yksi Haskell-ajokonteksti (engl. *Haskell Execution Context, HEC*). Haskell-ajokonteksti on tietorakenne, joka sisältää käyttöjärjestelmäsäikeen tarvitsemat tiedot Haskell-säikeen suorittamiseksi. Tämän tutkielman kannalta olennaisena ajokontekstiin kuuluvat seuraavat tiedot [MPS09]:

- Omistaja. Kertoo mikä käyttöjärjestelmäsäie suorittaa kontekstia tällä hetkellä.
- Viestijono. Pyynnöt muilta ajokonteksteilta.
- Suoritusjono. Suoritusta odottavat säikeet.
- Kipinävarasto. Kipinöidyt tyngät lisätään kipinöivän ajokontekstin kipinävarastoon.
- Käyttöjärjestelmäsäievarasto. Työtä vailla olevat käyttöjärjestelmäsäikeet.

Ajokontekstiin kuuluu lisäksi muita osia liittyen muistinhallintaan, säikeiden synkronointiin ja automaattiseen roskienkeruuseen. Jokainen aktiivinen ajokonteksti suorittaa töitä tärkeysjärjestyksen mukaan. Listassa alempana olevia suoritetaan vain, mikäli ylempänä listassa olevia tehtäviä ei ole tarjolla. Tärkeysjärjestys on seuraavanlainen [MPS09]:

1. Tarkista onko viestijonossa suoritusta odottavia pyyntöjä. Jos on, suorita ne.
2. Aja suoritusjonossa oleva säie. Säikeiden aikataulutuksen periaatteena on yksinkertainen kiertävä järjestys ilman prioriteetteja.
3. Jos kipinävarastossa on kipinöitä, luo kipinää vastaava säie ja suorita se.

4.2 Evaluoinnin tasot

Ennen varsinaisen Glasgow Parallel Haskellin tarkasteluun siirtymistä käydään vielä läpi ns. evaluoinnin tasot, joita käytetään ilmaisemaan kuinka paljon jostakin lausekkeesta evaluoidaan. Haskellin yhteydessä puhutaan yleensä kahdesta tasosta: normaalimuodosta (engl. *normal form*) ja heikosta normaalimuodosta (engl. *weak head normal form*). Lausekkeen sanotaan olevan normaalimuodossa silloin kun se on evaluoitu niin pitkälle kuin mahdollista, eli se ei sisällä lainkaan tynkiä [OGS08]. Esimerkiksi seuraavat lausekkeet ovat normaalimuodossa:

```
25
"hei"
\x → x*x
```

Seuraavat lausekkeet eivät ole normaalimuodossa:

```
-- Tässä voitaisiin vielä evaluoida summa
25+1
-- Tässä voitaisiin vielä evaluoida funktion arvo
(\x → x*x) 2
```

Heikossa normaalimuodossa lauseke on evaluoitu ulkoa katsottuna uloimpaan arvokonstruktoriin (ks. luku 3.8.2) tai lambdalausekkeeseen saakka. Esimerkiksi seuraavat lausekkeet ovat heikossa normaalimuodossa:

```
-- Uloin osa on arvokonstruktori(,)
("ab"++"cd","ef"++"gh")
-- Uloin osa on lambdalauseke
\x → x*x
```

Seuraavat lausekkeet taas eivät ole heikossa normaalimuodossa:

```
-- Uloin osa on funktion (++) arvon evaluointi
"ab"++"cd"
-- Uloin osa on summan 1+ (2+3) evaluointi
(1+(2+3))
```

Heikossa normaalimuodossa sisemmät alilausekkeet voivat olla joko evaluoituja tai eivät, sillä ei ole merkitystä. Näin ollen normaalimuodossa oleva lauseke on aina myös heikossa normaalimuodossa, mutta heikossa normaalimuodossa oleva lauseke ei välttämättä ole normaalimuodossa.

4.3 Operaattorit

Glasgow Parallel Haskell tarjoaa rinnakkaisuuden ilmaisemiseen 'par'-operaattorin, jonka tyypinmäärittely on muotoa:

```
par :: a → b → b
```

Tyypinmäärittelyn perusteella operaattori on siis vain projektio jälkimmäiseen parametriin. Lauseke 'par x y' evaluoi ja palauttaa arvon 'y', mutta ilmoittaa samalla ajoympäristölle, että ensimmäinen parametri 'x' voidaan evaluoida rinnakkain toisessa säikeessä samalla kun isäntäsäie jatkaa 'y':n evaluointia. Eli siis 'x' kipinöidään ja jatketaan 'y':n evaluointia. Tärkeää on huomata että kipinöinti ei välttämättä luo uutta säiettä, se ainoastaan ilmaisee mahdollisuuden siihen. Ajoympäristö voi tarpeen mukaan viivyttää tai jättää kokonaan huomiotta kipinöityjä lausekkeita. On hyvin tavallista, että suuresta osasta kipinöityjä lausekkeita ei koskaan luoda uutta säiettä. Yksittäisen tyngän kipinöinti on suhteellisen yksinkertainen ja halpa operaatio, jossa vain luodaan osoitin kipinöityyn tynkään ja sijoitetaan tämä osoitin prosessorin kipinävarastoon (engl. *spark pool*) [THL96].

Glasgow Parallel Haskell sisältää myös toisen operaattorin, 'pseq', joka ilmaisee eksplisiittisesti peräkkäissuoritusta. Sen tyypinmäärittely on identtinen 'par'-operaattorin kanssa:

```
pseq :: a → b → b
```

Toiminnallisesti 'pseq x y' evaluoi 'x':n heikkoon normaalimuotoon ja sen jälkeen evaluoi ja palauttaa 'y':n. 'Pseq'-operaattoria käytetään laskujärjestyksen määrittämiseen. Sen käyttö takaa laskujärjestyksen laiskassa ympäristössä, jossa laskujärjestys on tavallisesti tuntematon.

Aluksi 'pseq'-operaattorin tarkoitus voi vaikuttaa hämärältä. Nopeasti ajatellen seuraavan tunnetun Fibonaccin-lukuja rinnakkain laskevan funktion pitäisi toimia vain 'par'-operaattoria käyttäen:

```
parFib :: Int → Int
parFib n | n ≤ 1     = 1
```

```
        | otherwise  = par n1 (n1+n2+1)
          where
           n1 = parFib (n-1)
           n2 = parFib (n-2)
```

Jos ’n’ on suurempi kuin 1, ’parFib (n-1)’ kipinöidään ja isäntäsäie jatkaa evaluoimalla ’parFib (n-1) + parFib (n-2) + 1’. Tämä funktio antaa oikean tuloksen, mutta se ei sisällä käytännössä lainkaan rinnakkaissuoritusta. Ajettaessa ’n1’ kipinöidään, mutta heti perään isäntäsäie jatkaa myös saman lausekkeen arvioimista lausekkeessa ’n1 + n2 + 1’ ja näin ollen kipinä *haihtuu* (engl. *fizzle*), eikä sitä koskaan suoriteta rinnakkain [JMS09]. Lausekkeen muuttaminen muotoon ’n2 + n1 + 1’ ei ratkaise ongelmaa kuin satunnaisissa yksittäistapauksissa, sillä laiskassa kielessä yhteenlaskuoperaation operandien laskujärjestyksestä ei ole mitään takeita. Varmasti rinnakkain toimiva versio ylläolevasta funktiosta käyttää operaattoria ’pseq’ ilmaisemaan laskujärjestyksen eksplisiittisesti ajoympäristölle [THL98]:

```
parFib :: Int → Int
parFib n | n ≤ 1     = 1
         | otherwise  = par n1 (pseq n2 n1+n2+1)
           where
             n1 = parFib (n-1)
             n2 = parFib (n-2)
```

Nyt isäntäsäie evaluoi varmasti ’n2’:n ennen yhteenlaskua ja kipinöity tynkä ’n1’ voidaan evaluoida rinnakkain.

4.4 Strategiat

Ohjelman rinnakkaistaminen käyttäen ’par’ ja ’pseq’-operaattoreita on suhteellisen helppoa, kun ohjelmat ovat pieniä kuten kappaleen 4.3 esimerkit. Ohjelmien koon kasvaessa

varsinainen koodi alkaa kuitenkin täyttyä 'par'- ja 'pseq'-merkinnöistä ja varsinaisen algoritmin seuraaminen muuttuu vaikeaksi. Tässä kohtaa mukaan astuvat ns. strategiat (engl. *evaluation strategies*), jotka ovat Glasgow Parallel Haskellin avainkäsite. Toteutukseltaan strategiat ovat tavallisia korkeamman tason Haskell-funktioita. Strategiat eivät kuitenkaan tee lainkaan itse algoritmiin liittyvää evaluointia, ne vain määrittelevät ohjelman dynaamista käyttäytymistä. Juuri tämä onkin strategioiden pääajatus, ne erottavat dynaamisen käyttäytymisen määrittelyn varsinaisesta algoritmista, jolloin molemmat ovat selkeämpiä ja helpompia ymmärtää [THL98]. Strategian ideana on, että se palauttaa parametristaan uuden nostetun (ks. luku 3.10) version, johon on lisätty haluttu dynaaminen käyttäytyminen. Dynaamisen käyttäytymiseen kuuluvat seuraavat osa-alueet [MML10]:

- Rinnakkaisuuden hallinta, eli mitä säikeitä luodaan ja milloin.
- Evaluoinnin taso.
- Säikeiden rakeisuus, eli sopivan kokoisten säikeiden luominen. Säikeelle annettavan työmäärän on reilusti ylitettävä säikeen luomisen aiheuttamat resurssikustannukset, jotta säikeen luominen olisi kannattavaa.
- Paikallisuus. Osa säikeen tekemästä työstä on sen tarvitseman tiedon ja lopulta säikeen tuloksen välittäminen, joten joissakin tapauksissa säikeen luominen on kannattavaa vain jos sen tarvitsema tieto on paikallista.

Strategia siis palauttaa parametristaan uuden, nostetun version johon on sulautettu haluttu dynaaminen käyttäytyminen. Noston poistamiseksi (engl. *unlifting*) käytetään funktiota 'runEval':

```
data Eval a = Done a

type Strategy a = a → Eval a

runEval :: Eval a → a
runEval (Done a) = a
```

Tyyppi ’Eval’ on monadi, joten sitä voidaan käyttää laskujärjestyksen määrittämiseen. GpH määrittelee neljä perusstrategiaa, jotka evaluoivat parametrinsa normaalimuotoon, heikkoon normaalimuotoon, eivät ollenkaan tai kipinöivät parametrinsa evaluoitavaksi rinnakkain [MML10].

```
instance Monad Eval where
 return x = Done x
 Done x >>= k = k x

r0 :: Strategy a  -- Ei evaluoi mitään parametristaan,
r0 x = return x   -- palauttaa vain sen nostetun version

rseq :: Strategy a       -- Evaluoi parametrinsa heikkoon normaalimuotoon
rseq = x `pseq` return x -- ja palauttaa sen nostetun version

rdeepseq :: NFData a => Strategy a   -- Evaluoi parametrinsa
rdeepseq x = rnf x `pseq` return x   -- kokonaan ja palauttaa
                                     -- sen nostetun version

rpar :: Strategy a         -- Kipinöi parametrinsa ja
rpar x = x `par` return x  -- palauttaa sen nostetun version
```

Uusia strategioita kirjoittavan ohjelmoijan ei tarvitse enää käyttää ’par’ ja ’pseq’-operaattoreita, vaan voidaan käyttää ’Eval’-monadia ja ’rpar’ ja ’rseq’-funktioita, jotka nostavat rinnakkaisuuden ilmaisemisen abstraktiotasoa verrattuna ’par’-ja ’pseq’-operaattoreiden käyttämiseen [MML10].

Strategian erottamiseksi varsinaisesta algoritmista käytetään ’using’-funktiota, joka toteuttaa parametrina annetun strategian toisena parametrina annetulle lausekkeelle ja palauttaa sen jälkeen lausekkeen. Funktion ’using’ määrittely on muotoa:

```
using:: a → Strategy a → a
ex `using` strat = runEval (strat ex)
```

Strategioita voidaan ketjuttaa (engl. *function composition*) kuten muitakin funktiota. Strategioiden ketjuttaminen tapahtuu käyttäen funktiota 'dot', joka käyttäytyy kuten tavallinen funktioiden ketjutus.

```
dot :: Strategy a → Strategy a → Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

Perusstrategioiden, 'Eval'-monadin ja strategioiden ketjutuksen avulla voidaan luoda erilaisille tietotyypeille omia strategioita. Tällainen on esimerkiksi listan alkiot läpikäyvä ja jokaiselle alkiolle strategian toteuttava 'evalList'. Funktio 'evalList' ei itsessään määritä rinnakkaisuutta, mutta strategioiden ketjutusta käyttäen sen avulla on helppo määrittää strategia, joka suorittaa strategian kaikille listan alkioille rinnakkain [MML10].

```
evalList :: Strategy a → Strategy [a]
evalList s []     = return []
evalList s (x:xs) = do x' ← s x
                       xs' ← evalList s xs
                       return (x' : xs')


parList :: Strategy a → Strategy [a]
parList s = evalList (rpar `dot` s)
```

4.5 Esimerkkejä

Strategioita voidaan käyttää sekä tietorinnakkaisuutta sisältävissä että tehtävärinnakkaisuutta sisältävissä ongelmissa. Quicksort on tehokas lajittelualgoritmi ja yksinkertainen kirjoittaa Haskellilla, ja sen strategioita hyödyntävä versio on lähes yhtä yksinkertainen. Quicksort-esimerkkiä käytetään myös luvussa 5.3 Data Parallel Haskellin yhteydessä. Quicksort on esimerkki tehtävärinnakkaisuudesta.

```
qsort :: Ord a ⇒ [a] → [a]
qsort [] = []
```

```
qsort (x:xs) = runEval (do low ← rpar (qsort (filter (< x) xs))
                           hi ← rseq (qsort (filter (≥ x) xs))
                           return (low ++ [x] ++ hi))
```

Tämä quicksort-esimerkki ei käytä lainkaan 'using'-funktiota strategioiden soveltamiseen, vaan funktioita 'rpar' ja 'rseq' on käytetty varsinaisen algoritmin seassa. Tavallaan tämä rikkoo strategioiden alkuperäistä periaatetta varsinaisen algoritmin ja dynaamisen käyttäytymisen erottamisesta. Kuitenkin käytetyn monadien do-notaation avulla tästä ilmenee selkeästi, että pienempien lajittelu kipinöidään suoritettavaksi rinnakkain samalla kun aloitetaan suurempien lajittelu ja lopuksi liitetään tuloslistat yhteen. Usein tavallisessa Haskellissa käytetään kirjastofunktiota 'partition' jakamaan lista 'low'-ja 'hi' -osiin, koska tällöin lista saadaan jaettua yhdellä läpikäynnillä. Tässä rinnakkaisversiossa 'partition'-funktion käyttö on kuitenkin huono idea, koska tällöin listan jakoa pienempiin ja suurempiin alkioihin ei voida tehdä rinnakkain. Algoritmiin muodostuu peräkkäin suoritettava pullonkaula, joka haittaa huomattavasti rinnakkaisuuden hyödyntämistä.

Esimerkiksi tietorinnakkaisuudesta ja strategioiden käytöstä sopii 'parMap', joka toimii kuten 'map', mutta käy listan läpi ja suorittaa evaluoinnin rinnakkain. Tietorinnakkaisissa ohjelmissa dynaaminen käyttäytyminen voidaan helposti erottaa algoritmista 'using'-funktiota käyttämällä.

```
parMap :: Strategy a → (a → b) → [a] → [b]
parMap strat func ls = map func ls `using` parList strat
```

5 DATA PARALLEL HASKELL

Data Parallel Haskell (DPH) on perinteisen Haskellin lisäosa, jolla voidaan toteuttaa implisiittisesti tietorinnakkaisia ohjelmia. Perinteisesti tietorinnakkaiset sovellukset ovat hyödyntäneet ainoastaan ns. yksitasoista rinnakkaisuutta (engl. *flat parallelism*), jossa rinnakkain käsiteltävän tietorakenteen (taulukko, lista tms.) alkiot ovat yksinkertaisia tietotyyppejä, kuten kokonaislukuja tai merkkejä. Data Parallel Haskell kuitenkin laajentaa tätä perinteistä mallia sallien myös ns. sisäkkäisen rinnakkaisuuden (engl. *nested data parallelism*), jossa rinnakkainen tietorakenne voi edelleen sisältää toisia rinnakkaisia tietorakenteita ja/tai operaatioita. DPH pohjautuu Alan Blellochin yhdeksänkymmentäluvun alkupuolella työtovereidensa kanssa kehittämään NESL-ohjelmointikieleen [BCH94], jossa sisäkkäiset rinnakkaiset tietorakenteet muunnetaan systemaattisesti useiksi yksitasoiksi rakenteiksi. Tämä sama muunnos on myös Data Parallel Haskellin avainidea, mutta NESLin alkuperäistä mallia on myös kehitetty mm. tukemaan käyttäjän itse määrittelemiä tyyppejä [CLJ07].

5.1 Operaattorit

Data Parallel Haskellissa rinnakkaisuus on implisiittistä. Rinnakkaisuuden ilmaisemiseen käytetään uutta rinnakkaista taulukkotyyppiä (engl. *parallel array*) ja sen operaatioita. Rinnakkaisten taulukkojen notaatio on hyvin samankaltainen kuin Haskellin tavallisten listojen. Esimerkiksi $[: Int :]$ on rinnakkainen kokonaislukutaulukko ja $[: String :]$ on rinnakkainen merkkijonotaulukko. Rinnakkaiset listat ovat semanttiselta käyttötarkoitukseltaankin hyvin samanlaisia (ja jopa identtisiä) kuin tavalliset listat, vaikka sisäisesti näiden kahden toteutukset poikkeavat hyvin paljon toisistaan. Kääntäjä jakaa rinnakkaisten listojen sisältämän datan ja siihen kohdistettavat operaatiot säikeisiin ja jakaa säikeet prosessointielementeille automaattisesti [CLJ07].

Rinnakkaiset taulukot ja tavalliset listat ovat sisäisesti täysin eri tyyppejä, joten rinnak-

kaisilla taulukoilla operoitaessa ei voida käyttää samoja kirjastofunktioita kuin tavallisilla listoilla. Data Parallel Haskell määrittelee rinnakkaisille taulukoille omat funktiot, jotka vastaavat toiminnallisuudeltaan tavallisten listojen kirjastofunktioita. Esimerkkejä rinnakkaisten taulukoiden kirjastofunktioista ovat mm:

```
filterP :: (a → Bool) → [:a:] → [:a:]
lengthP :: [:a:] → Int
mapP    :: (a → b) → [:a:] → [:b:]
zipP    :: [:a:] → [:b:] → [:(a,b):]
```

Rinnakkaisten taulukoiden funktiot on lähes kaikki nimetty alkuperäisten listafunktioiden mukaan lisäten vain loppuun kirjain P. Rinnakkaiset taulukot tukevat myös taulukkokonstruktoreita, jotka ovat syntaksiltaan identtisiä tavallisten listakonstruktoreiden kanssa.

```
lessThanSeven :: [:Int:] → [:Int:]
lesThanSeven xs = [: x | x ← xs, x < 7 :]
```

Toisin kuin tavallinen lista, rinnakkainen taulukko ei ole laiska tietorakenne, vaan taulukon yhden elementin kutsuminen aiheuttaa kaikkien taulukon alkioiden evaluoimisen. Tavallisista listoista tuttuja äärettömiä listarakenteita ei siis voida käyttää. Toinen merkittävä rajoite rinnakkaisten taulukoiden käytössä on, että vielä toistaiseksi ne eivät tue tyyppiluokkia, vaan kullekin tyypille on kirjoitettava oma erikoistunut funktionsa [PCL08].

5.2 Sisäkkäisen rinnakkaisuuden eliminointi

Data Parallel Haskellin avainidea on NESL-kielessä alun perin esitelty kääntäjän suorittama systemaattinen muunnos, jossa sisäkkäinen rinnakkaisuus eliminoidaan ohjelmasta muuntamalla sisäkkäinen rinnakkaisuus litteäksi rinnakkaisuudeksi. Haskellin tyyppijärjestelmä on huomattavasti laajempi kuin NESLin vastaava, joten myös muunnos on monimutkaisempi. Tämän muunnoksen, jota kutsutaan vektorisaatioksi, lisäksi kääntäjä luo automaattisesti suoritettavat säikeet ja jakaa laskentakuorman niiden kesken mahdollisimman

tasaisesti. Kokonaisuudessaan Data Parallel Haskell toteuttaa implisiittisen rinnakkaisuuden käännösvaiheen neljällä toimenpiteellä, jotka ovat listakonstruktoreiden eliminointi, vektorisaatio, välitulosten eliminointi (optimointi) sekä säikeisiin jako [PCL08].

5.2.1 Listakonstruktoreiden eliminointi

Aivan kuten tavallisten listakonstruktoreiden tapauksessakin, myös rinnakkaisten listojen tapauksessa listakonstruktorit ovat vain apukeinoja, joilla käyttäjän on helpompi luoda uusia listoja. Käännösvaiheessa listat puretaan listoilla operoivien funktioiden kutsuiksi. Rinnakkaiset listojen konstruktorit puretaan lähes täysin samoin säännöin kuin tavalliset listakonstruktorit. Listakonstruktorit ovat yleisesti syntaksiltaan seuraavan muotoisia [Pey12]:

`[: e | q1,q2,...qn :]`, missä $n \geq 1$ ja `qi` on jokin seuraavista:

```
x ← y          (muodostaja)
let z = ...    (vakion määrittely)
p                (predikaatti)
```

Muodostajassa 'y' on lista, jonka arvoista 'x' muodostetaan. Vakion määrittelyssä määritellään uusi paikallinen vakio, jota voidaan käyttää predikaateissa ja muodostajissa. Predikaatti on mikä tahansa Bool-tyyppinen lauseke. Tämän syntaksin puitteissa listakonstruktorit puretaan listoiksi ja funktiokutsuiksi seuraavien ehtojen mukaan:

```
[:e | True:]          = [:e:]
[:e | qs :]           = [:e | qs, True]
[:e | p,qs:]          = if p then [:e | qs :] else [::]
[:e | x ← ys, qs:]    = let ok x = [:e | qs :]
                            ok _ = [::]
                        in concatMap ok ys

[:e | let vars, qs :] = let vars in [:e | qs :]
```

missä ’e’ on jokin lauseke, ’x’ on jokin hahmo, ’ys’ on lista, ’vars’ yksi tai useampi vakio, ’p’ on edelleen predikaatti ja ’qs’ yksi tai useampi muodostaja, vakio tai predikaatti.

5.2.2 Vektorisaatio

Vektorisaation tarkoituksena on eliminoida ohjelmasta sisäkkäiset tietorinnakkaiset rakenteet. Tarkoituksena on, että lopullinen suoritettava ohjelma sisältää vain litteää tietorinnakkaisuutta.

Vektorisaation avainideana on, että ohjelman jokaisesta funktiosta muodostetaan nostettu versio, joka operoi rinnakkaisilla listoilla. Esimerkiksi seuraava yksinkertainen funktio nostaa parametrinsa toiseen potenssiin ja lisää tähän arvoon yhden.

```
func :: Float → Float
func x = x*x + 1
```

Kääntäjä muodostaa käännöksen aikana erityisten sääntöjen mukaan kaikista ohjelman sisältämistä funktioista rinnakkaisilla listoilla operoivat nostetut versiot. Edellisen esimerkkifunktion nostettu versio on seuraavan muotoinen:

```
funcL :: [:Float:] → [:Float:]
funcL x = (x *L x) +L (replicateP n 1)
          where
           n = lengthP x
```

Tässä määritelmässä kaikki funktiossa ’func’ esiintyneet funktiot on korvattu nostetuilla versioillaan, eli ’func’ on korvattu ’funcL’:llä, ’*’ on korvattu ’*L’:llä jne. Alkuperäisen funktion määritelmän vakio 1 on korvattu pelkästään tätä vakiota sisältävällä listalla.

Yleisesti kaikki nostetut funktiot noudattavat kaavaa ’funcL = mapP func’. Vektorisaatiossa kaikki kutsut ’mapP func’ korvataan tällä nostetulla funktiolla ’funcL’ [PCL08]. Alan Blellochin ja muiden NESL-ohjelmointikielen kehittäjien oivallus oli, että nostetun funktion nostettu muoto voidaan ilmaista alkuperäisen nostetun funktion avulla [BlS90]. Tä-

män muunnoksen avulla mielivaltaisen syvätkin sisäkkäiset rinnakkaiset rakenteet saadaan litistettyä perättäisiksi operaatioiksi rinnakkaisille listoille.

Yllä on esitetty vektorisaation yksinkertainen avainidea. Käytännössä vektorisaatio on kuitenkin erittäinen monimutkainen ja monivaiheinen muunnos. Funktioiden nostamisen lisäksi vektorisaatiossa rinnakkaiset listat muunnetaan algebrallisiksi tyypeiksi ja sisäkkäiset listat muunnetaan tämän tyypin avulla edelleen useaksi yksinkertaiseksi listaksi. Ohjelmoijan itse määrittämät tietotyypit muunnetaan edelleen nostetuiksi versioiksi funktioiden tapaan [PCL08].

5.2.3 Välitulosten eliminointi

Vektorisaatiomuunnoksen tuottama koodi sisältää paljon muistia kuluttavia väliaikaisia listoja sekä turhia funktiokutsuja. Välitulosten eliminoinnissa (engl. *fusion*) nämä ylimääräiset osat pyritään eliminoimaan, jotta lopullinen ohjelmakoodi olisi mahdollisimman tehokasta.

Välitulosten eliminoinnin ensimmäinen vaihe on turhien funktiokutsujen poistaminen. Ohjelmakoodin vektorisaatio suoritetaan systemaattisten sääntöjen mukaan, ja tässä muunnoksessa syntyy todella paljon funktiokutsuja, jotka kumoavat toisensa. Kääntäjä eliminoi tiettyjen sääntöjen perusteella systemaattisesti tällaiset turhat funktiokutsut, joilla ei ole merkitystä isäntäfunktion lopullisen tuloksen kannalta.

Toisessa vaiheessa kääntäjä eliminoi monimutkaisen prosessin ja systemaattisten uudelleenkirjoitussääntöjen avulla vektorisaatiossa syntyneet väliaikaiset listat ohjelmakoodista. GHC tekee välitulosten eliminointia myös tavalliselle Haskell-koodille, varsinkin kun käytetään listoja ja niiden operaatioita. Data Parallel Haskellia varten eliminointia on muokattu ottamaan huomioon rinnakkaiset taulukot ja niiden operaatiot. Tämä tekee välitulosten eliminoinnista jonkin verran mutkikkaamman prosessin kuin tavallisen Haskell-koodin tapauksessa [CLJ07].

5.2.4 Säikeisiin jako

Viimeisenä vaiheena muunnoksessa on rinnakkaisten listojen ja niillä operoivien funktioiden jako sopivan kokoisiin osiin ja näiden osien jakaminen suoritettavaksi säikeille. Yleensä säikeitä luodaan yksi jokaista käytettävissä olevaa prosessointielementtiä kohden. Säikeiden luomiseen ja niiden väliseen kommunikointiin käytetään Concurrent Haskellin kirjastofunktioita.

Säikeiden välisen työn jako ja niiden välinen kommunikointi tehdään eksplisiittisesti tyypin 'Dist a' avulla. Esimerkiksi 'Dist [:Float:]' kuvaa rinnakkaisen listan osia (yksi per säie), jotka muodostavat koko listan. Rinnakkaiset listat jaetaan säikeisiin ja kootaan yhteen kahden funktion avulla:

```
splitD :: PA a → Dist (PA a)
joinD  :: Dist (PA a) → PA a
```

missä 'PA' on rinnakkaisesta listasta vektorisaatiossa muunnettu algebrallinen tietotyyppi, kuten kohdassa 5.2.2 mainittiin. Näiden funktioiden kutsut koodissa merkitsevät säikeiden välistä kommunikointia, kun taas 'mapD'-funktion kutsu merkitsee säikeiden itsenäisesti suorittamaa prosessointia.

```
mapD :: (a → b) → Dist a → Dist b
```

Vektorisaation jälkeen ohjelmakoodi ei sisällä sisäkkäisiä rinnakkaisia rakenteita, joten 'mapD' suorittaa puhtaasti perättäisen funktion sille osoitetun listan osalle, eli semantiikaltaan se muistuttaa suuresti tavallista 'map'-funktiota, vaikka rinnakkaisen listan esitystapa on kääntäjän näkökulmasta täysin erilainen kuin tavallisen listan.

5.3 Esimerkkejä

Data Parallel Haskellin perimmäinen tarkoitus on, että ohjelmoija voi kirjoittaa ohjelmansa aivan kuten perättäisohjelman, hyödyntäen vain rinnakkaisia taulukoita listojen sijaan.

Seuraavassa esimerkissä Haskell-quicksort on kirjoitettu käyttäen rinnakkaisia taulukoita:

```
qsort :: Ord a ⇒ [:a:] → [:a:]
qsort [::] = [::]
qsort (x:xs) = lesser ++ [:x:] ++ bigger
  where
    lesser = qsort [: y | y ← xs, y < x:]
    bigger = qsort [: y | y ← xs, y ≥ x:]
```

Toistaiseksi tämä esimerkki on kuitenkin vasta tavoitetila, sillä Data Parallel Haskellin kehitys on pahasti kesken. Yllä oleva koodia ei voi tällä hetkellä kääntää sellaisenaan, vaan Data Parallel Haskellin käytössä on suuria rajoituksia. Esimerkiksi yksi moduuli ei voi sisältää vektorisoitua ja tavallista Haskell-koodia, vaan nämä pitää erotella omiin moduuleihinsa. Lisäksi tyyppiluokkia ei voi käyttää vektorisoidussa koodissa.

Kirjoitushetkellä minkäänlaisen DPH-ohjelman kääntäminen ei onnistunut 64-bittisessä Windows 7- tai 32-bittisessä Ubuntu Linux ympäristössä. Virallisen dokumentaation mukaan DPH-ohjelmien pitäisi olla käännettävissä tietyin erityisehdoin, mutta useista yrityksistä huolimatta tämä ei onnistunut, johtuen ilmeisesti DPH-kirjastojen dokumentoimattomista riippuvuuksista.

5.4 Repa-kirjasto

Repa-kirjasto on Data Parallel Haskellin kanssa samankaltainen, mutta huomattavasti valmiimpi tekniikka implisiittisesti tietorinnakkaisten ohjelmien toteuttamiseen. Repa on lyhenne sanoista Regular Parallel Arrays. Repa-kirjasto esittelee ohjelmoijan käyttöön moniulotteiset, rinnakkaiset taulukot samaan tapaan kuin Data Parallel Haskell. Taulukot ovat ulottuvuuksien suhteen polymorfisia (engl. *shape polymorphism*), eli samoja funktioita, kuten 'map' ja 'sum', voidaan käyttää riippumatta siitä montako ulottuvuutta käsiteltävässä taulukossa on [KCL10].

Kirjastolla on samat kehittäjät kuin Data Parallel Haskellilla ja Repa käyttääkin paljon Data Parallel Haskellia varten kehitettyjä rakenteita taulukoiden sisäiseen esittämiseen. Repa-kirjaston taulukoiden käyttö poikkeaa kuitenkin huomattavasti Data Parallel Haskellin taulukoiden tai tavallisten listojen käytöstä, ja vaatii ohjelmoijalta jonkin verran perehtymistä ja erityishuomiota. Repa-ohjelmien rinnakkaisuuden optimointi vaatii myös ohjelmoijalta huomattavaa perehtymistä eikä selkeitä parhaita käytäntöjä ole (ainakaan toistaiseksi) olemassa. Näistä syistä johtuen Repa-kirjaston tarkempi tarkastelu rajattiin tämän tutkielman ulkopuolelle.

6 TULOKSET

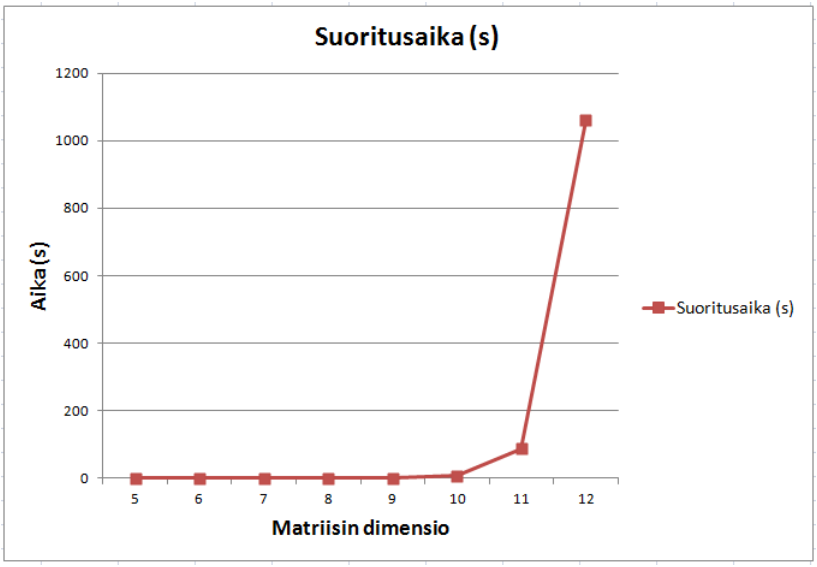
Tässä luvussa tarkastellaan pienen esimerkkiohjelman avulla kuinka tavallisen Haskell-ohjelman rinnakkaistaminen onnistuu Glasgow Parallel Haskellin avulla ja minkälainen parannus suorituskykyyn saadaan ohjelmoijan kohtuullisella lisäpanostuksella. Ensin tarkastellaan kuinka tavalllinen, täysin perättäinen ohjelma suoriutuu testidatan käsittelystä. Tämän jälkeen esitellään kolme erilaista versiota samasta ohjelmasta, jotka kaikki hyödyntävät luvussa 4 esiteltyjä funktioita, ja tarkastellaan miten nämä ohjelmat suoriutuvat testidatan käsittelystä kun käytössä on 2–4 prosessoria.

Esimerkkiohjelma lukee matriiseja käyttäjän määrittämästä tiedostosta, laskee matriisien determinantit ja kirjoittaa ne toiseen käyttäjän määrittämään tiedostoon. Matriisit esitetään kaksinkertaisena listana liukulukuja (eli listana joka sisältää listoja) ja matriisit lasketaan Laplace-kehitelmän avulla, jossa matriisin determinantti lasketaan sen alimatriisien determinanttien summana.

Liitteessä A on esitetty ohjelman alkuperäinen versio, joka on tavanomainen Haskell-ohjelma eikä sisällä minkäänlaista rinnakkaisuutta. Ohjelmalle syötetään ensin 10 000 pientä matriisia joiden dimensiot ovat satunnaisesti välillä 3–8. Kaikki matriisit ovat neliömatriiseja. Matriisien sisältämä data on kaikissa tapauksissa satunnaisia desimaalilukuja väliltä 1–10 000 kahden desimaalin tarkkuudella esitettynä.

Toisena aineistona ohjelmalle syötetään 25 kpl 10×10 matriiseja, sisältäen edelleen satunnaisia desimaalilukuja välillä 1–10 000. Kokeet suuremmilla matriiseilla olisivat informatiivisempia, mutta matriisit joiden dimensiot ovat yli 10 osoittautuivat ohjelmalle liian raskaiksi laskettavaksi alle 5 minuutin ajassa. Tämä johtuu käytetyn algoritmin eksponentiaalisesta aikakompleksisuudesta, eli algoritmin suoritusaika kasvaa eksponentiaalisesti suhteessa syötteen koon kasvuun. Pienemmillä dimensioilla laskenta on vielä melko nopeaa, mutta suoritusaika kasvaa äkisti kun matriisin dimensio lähestyy kymmentä. Kuvassa 3 on esitetty ohjelman suoritusaika yhden matriisin determinanttia laskettaessa. Kuvasta käy selkeästi ilmi algoritmin eksponentiaalinen aikakompleksisuus. Kun matriisin dimen-

sio on viisi aikaa kului 0.03 sekuntia. Kun matriisin dimensiota kasvatettiin kahteentoista suoritusaika oli jo 1060 sekuntia eli lähes 18 minuuttia. Tästä syystä toisena aineistona päädyttiin käyttämään 10×10 matriiseja, joiden determinantit esimerkkiohjelma pystyy vielä laskemaan kohtuullisessa ajassa.



Kuva 3: Yhden matriisin determinantin laskemiseen kulunut aika matriisin dimensioilla 5-12.

Determinanttien laskemiseen on olemassa huomattavasti parempiakin algoritmeja, mutta tämä versio on vielä suhteellisen helppo ymmärtää ja toteuttaa. Tutkielman kannalta tärkeintä on, että ohjelma sisältää raskasta laskentaa, joka mahdollisesti hyötyisi rinnakkaistamisesta. Tarkoituksena ei ole myöskään etsiä kaikkein optimaalisinta rinnakkaissuoritusta vaan tutkia minkälaisia muutoksia ohjelmakoodiin ja minkälainen ohjelmoijan lisäpanos vaaditaan, jotta saavutetaan kohtalainen parannus suoritusajassa.

Kaikki ohjelman koeajot on tehty kolmeen kertaan ja tässä kappaleessa esitetyt tulokset ovat noiden ajojen keskiarvoja. Kaikki ohjelman versiot on käännetty Glasgow Haskell Compilerin versiolla 7.0.3. Rinnakkaisversioissa on käytetty käännösvaiheen -threaded parametria, jolla ohjelma käännetään monisäikeiseksi. Ajovaiheessa käytettävien prosessorien määrä on määritetty käyttäen GHC:n -N parametria, ja yksittäisen ajon statistiikka on saatu ulos käyttämällä ajovaiheessa GHC:n -s parametria. Testikoneen käyttöjärjestelmänä

oli 64-bittinen Windows 7 Professional, suorittimena neliytiminen 3 GHz AMD Phenom II X4 940. Jokaisella prosessoriytimellä oli 512 KB L2-välimuistia ja koko prosessorilla 6 MB L3-välimuistia. Testikoneessa RAM-muistia oli 6 GB.

Tässä kappaleessa on esitetty vain ohjelmien suoritusten tulokset ja kerrottu lyhyesti mikä oli pääajatus kunkin version kehityksen takana. Tulosten merkitystä on pohdittu tarkemmin kappaleessa 7.

6.1 Perättäisversio

Liitteessä A on esitelty Haskell-ohjelma (Determinants.hs), joka lukee komentoriviparametrina saamastaan tiedostosta matriiseja, laskee kunkin matriisin determinantin ja kirjoittaa ne toisena parametrina saamaansa tekstitiedostoon. Ohjelma ei sisällä minkäänlaista rinnakkaisuutta. Taulukossa 1 on esitetty ohjelman suoritukseen kulunut aika ja käytetyn muistin määrä käyttäen kappaleessa 6 mainittuja aineistoja.

Taulukko 1: Suoritusajat ja käytetty muisti Determinants-ohjelman perättäisversiolla.

Pienemmät matriisit, 10000 $\times$ (3-8) $\times$ (3-8)	
Suoritusaika (s)	Käytetty muisti (MB)
32.08	2
Suuremmat matriisit, 25 $\times$ 10 $\times$ 10	
Suoritusaika (s)	Käytetty muisti (MB)
205.17	2

Dimensioltaan suurempien matriisien determinanttien laskeminen vei huomattavasti kauemmin kuin pienempien matriisien, vaikka pienempiä matriiseja oli käsiteltävänä satoja kertoja enemmän. Kuten edellä jo mainittiin, tämä johtuu algoritmin eksponentiaalisesta aikakompleksisuudesta. Muistia ohjelma käytti saman verran molemmilla aineistoilla. Tämä tilanne toimi pohjana, kun ohjelmasta ryhdyttiin kehittämään rinnakkaislaskentaa hyödyntävää versiota, joka suoriutuisi determinanttien laskemisesta nopeammin.

6.2 Ensimmäinen rinnakkaisversio

Determinants -ohjelman rinnakkaisversion kehityksessä ensimmäinen ajatus oli mahdollisimman aggressiivinen rinnakkaislaskennan hyödyntäminen. Tämä strategioita mahdollisimman laajalla alueella hyödyntävä ohjelma on esitetty liitteessä B. Rinnakkaisuutta on pyritty hyödyntämään mahdollisimman paljon ohjelman suorituksen jokaisessa vaiheessa. Tämän ohjelman suorituksen tulokset on esitetty taulukossa 2.

Taulukko 2: Suoritusajat ja käytetty muisti Determinants-ohjelman ensimmäisellä rinnakkaisversiolla. Suoritusaikoja on vertailtu ohjelman perättäisversioon.

Pienemmät matriisit, 10000 $\times$ (3-8) $\times$ (3-8)			
Prosessorit (kpl)	Suoritusaika (s)	% edellisestä versiosta	Käytetty muisti (MB)
2	40.73	127%	87
3	30.09	98%	90
4	26.05	81%	92
Suuremmat matriisit, 25 $\times$ 10 $\times$ 10			
Prosessorit (kpl)	Suoritusaika (s)	% edellisestä versiosta	Käytetty muisti (MB)
2	263.28	128%	3
3	187.17	91%	4
4	162.75	79%	4

Kahdella prosessorilla suoritettuna suoritusaika on jopa huonompi kuin perättäisversiolla vastaavalla aineistolla. Huomiota herätti myös pienempien matriisien aineistolla huomattavan suuri lisäys käytetyn muistin määrässä.

6.3 Toinen rinnakkaisversio

Ensimmäinen yritys ohjelman suorituskyvyn parantamiseksi tuotti melko vaatimattoman tuloksen. Ensimmäinen versio kipinöi lausekkeita ohjelman jokaisessa vaiheessa, joten seuraava askel olikin yrittää karsia rinnakkaisuutta ohjelmasta. Rinnakkaisuutta karsittiin pois apufunktioista, jotka ovat nopeita suorittaa perättäinkin eivätkä suorita raskasta laskentaa,

joten ne eivät välttämättä hyötyisi lisäsäikeiden luonnista. Tämän version koodi on esitetty liitteessä C. Tämän version suorituksen tulokset on esitetty taulukossa 3.

Taulukko 3: Suoritusajat ja käytetty muisti Determinants-ohjelman toisella rinnakkaisversiolla. Suoritusaikoja on vertailtu ensimmäiseen rinnakkaisversioon.

Pienemmät matriisit, 10000 $\times$ (3-8) $\times$ (3-8)			
Prosessorit (kpl)	Suoritusaika (s)	% edellisestä versiosta	Käytetty muisti (MB)
2	26.22	64%	66
3	18.85	63%	71
4	15.88	61%	67
Suuremmat matriisit, 25 $\times$ 10 $\times$ 10			
Prosessorit (kpl)	Suoritusaika (s)	% edellisestä versiosta	Käytetty muisti (MB)
2	150.76	57%	3
3	106.12	57%	4
4	90.79	56%	4

Ohjelman suorituskyky parani huomattavasti karsimalla rinnakkaisuutta. Suoritusajat lähes puolittuivat ensimmäiseen rinnakkaisversioon verrattuina ja ovat jo selkeästi parempia kuin ohjelman perättäisversiolla. Pienempien matriisien aineistolla muistia käytetään edelleen huomattavasti enemmän kuin perättäisversiossa, mutta toisaalta muistin käyttö tippui paljon ensimmäiseen rinnakkaisversioon verrattuna.

6.4 Kolmas rinnakkaisversio

Kolmannessa versiossa jatkettiin samalla tiellä kuin toisessa versiossa, eli rinnakkaisuutta karsittiin edelleen. Kolmannessa versiossa rinnakkaisuutta hyödynnetään vain kokonaisten matriisien determinanttien laskennan välillä. Yksittäisen matriisin determinantin laskeminen tapahtuu täsmälleen samoin kuin perättäisversiossa. Tämän ohjelman suorituksen tulokset on esitetty taulukossa 4.

Taulukko 4: Suoritusajat ja käytetty muisti Determinants-ohjelman kolmannella rinnakkaisversiolla. Suoritusaikoja on vertailtu toiseen rinnakkaisversioon.

Pienemmät matriisit, 10000 × (3-8) × (3-8)			
Prosessorit (kpl)	Suoritusaika (s)	% edellisestä versiosta	Käytetty muisti (MB)
2	22.14	84%	71
3	15.61	83%	70
4	12.56	79%	66
Suuremmat matriisit, 25 × 10 × 10			
Prosessorit (kpl)	Suoritusaika (s)	% edellisestä versiosta	Käytetty muisti (MB)
2	108.25	72%	3
3	76.75	72%	3
4	61.94	68%	4

Tämän ohjelman suorituskyky on edelleen huomattavasti parempi kuin perättäisversion ja myös parempi kuin kummankaan aiemman rinnakkaisversion. Suorituskyvyn parannus verrattuna toiseen rinnakkaisversioon ei kuitenkaan enää ollut yhtä suuri kuin toisen version parannus verrattuna ensimmäiseen versioon. Mielenkiintoisesti muistin kulutus ei kuitenkaan pienentynyt toiseen rinnakkaisversioon verrattuna kuten olisi saattanut odottaa. Tämäkin ohjelman versio käyttää muistia huomattavasti enemmän kuin perättäisversio.

Viimeisestä ja parhaan suorituskyvyn versiosta voidaan laskea kappaleessa 2.2.2 esitetyn Amdahlin lain avulla, kuinka suuri osa kolmannen rinnakkaisversion koodista on suoritettavissa rinnakkain. Otetaan esimerkiksi pienempien matriisien aineisto neljällä prosessorilla suoritettuna, jonka suoritusaika oli 12.56 sekuntia. Perättäisversiolla saman aineiston laskemiseen kului 32.08 sekuntia. Tällöin ohjelman rinnakkaisen version nopeutuminen suhteessa alkuperäiseen on $\frac{32.08}{12.56} = 2.55 = S$. Toisaalta Amdahlin lain mukaan $S = \frac{1}{(1-P)+\frac{P}{N}}$. Tässä tapauksessa prosessoreiden määrä N on 4, joten ratkaisemalla yhtälö P:n suhteen saadaan:

$$P = \frac{\frac{1}{S} - 1}{\frac{1}{N} - 1} = 0.81 = 81\% \tag{3}$$

Pienemmän aineiston tapauksessa siis 81% ohjelmasta olisi rinnakkain suoritettavissa. Jos

sama laskutoimitus tehdään suurten matriisien tapaukselle neljällä prosessorilla suoritettuna, saadaan $S = \frac{205.17}{61.94} = 3.31$. Kun tämä sijoitetaan kaavaan (3) saadaan $P = 93\%$. Näyttäisi siis, että suurempien matriisien aineistolla isompi osa ohjelmasta voidaan suorittaa rinnakkain. Suoritusaikojen vertailu vastaavasti myös kahden ja kolmen prosessorin tapauksissa vahvistaa tämän tuloksen. Tulos tuntuu loogiselta, koska pienempien matriisien aineistossa matriiseja, ja näin ollen myös säikeitä, on huomattavasti enemmän. Tällöin myös säikeiden koordinointia on enemmän. Samoin pienempien matriisien aineistolla ohjelma sisältää huomattavasti enemmän tiedostosta lukemista ja tiedostoon kirjoitusta, jotka molemmat tapahtuvat perättäissuorituksena.

7 YHTEENVETO JA POHDINTA

Perinteisesti ohjelmoitaessa rinnakkaisia järjestelmiä ohjelmoijan on huolehdittava varsinaisen algoritmin oikeellisuudesta ja tämän lisäksi rinnakkaisuuden koordinoinnista. Suurien rinnakkaisten ohjelmien kehittäminen perinteisillä proseduraalisilla kielillä on hyvin työlästä ja virhealtista. Uusia ratkaisuja rinnakkaisuuden toteuttamisen helpottamiseksi kehitetään jatkuvasti. Yhtenä lupaava ehdokkaana rinnakkaisuuden automatisointiin pidetään funktionaalisia ohjelmointikieliä. Tässä tutkielmassa tarkasteltiin rinnakkaisuuden toteutusta Haskellissa, joka on puhdas, funktionaalinen ohjelmointikieli. Puhtaassa kielessä funktiot eivät sisällä lainkaan sivuvaikutuksia, jonka vuoksi rinnakkaisuuden automatisointia on helpompi toteuttaa.

Tutkielman toisessa luvussa käsiteltiin rinnakkaisuutta yleisesti, sekä selvitettiin mikä erottaa rinnakkaisuuden samanaikaisuudesta. Toisessa luvussa esiteltiin myös rinnakkaisuuteen liittyviä käsitteitä, kuten tietorinnakkaisuus, tehtävärinnakkaisuus ja säikeet. Lisäksi esiteltiin Amdahlin ja Gustafssonin lait, jotka kuvaavat kuinka paljon ohjelman suoritusta voidaan teoreettisesti nopeuttaa rinnakkaislaskentaa hyödyntämällä.

Luvussa kolme käsiteltiin yleisesti Haskellin historiaa ja esiteltiin kaksi Haskelliin läheisesti liittyvää käsitettä: puhtaus ja laiska evaluointi. Puhtaus tarkoittaa sivuvaikutusten puuttumista ja laiska evaluointi termien evaluoinnin viivyttämistä siihen asti, että niitä oikeasti tarvitaan. Kolmannen luvun lopuksi käsiteltiin Haskell-ohjelmoinnin keskeisimpiä käsitteitä, kuten funktiot, listat, omat tietotyypit ja monadit.

Neljännessä luvussa esiteltiin Glasgow Parallel Haskell, joka on Haskellin rinnakkaissuoritukseen tarkoitettu lisäkirjasto. Glasgow Parallel Haskell sisältää kaksi perusoperaattoria rinnakkaisuuden ilmaisemiseen, ’par’ ja ’pseq’, joiden käyttöä myös tarkasteltiin neljännessä luvussa. Lisäksi tarkasteltiin strategioita, jotka ovat korkeamman tason rakenteita rinnakkaisuuden toteuttamiseen ja hyödyntävät kahta em. perusoperaattoria.

Viidennessä luvussa käsiteltiin Data Parallel Haskellia, joka on perinteisen Haskellin lisä-

kirjasto tietorinnakkaisten ohjelmien toteutukseen. Data Parallel Haskell toteuttaa rinnakkaisuuden implisiittisesti, eli ohjelmoija kirjoittaa ohjelman kuten tavallisesti, ja ajoympäristö huolehtii rinnakkaisuuden käytännön toteutuksesta. Data Parallel Haskellin kehitystyö oli vielä tutkielman kirjoitushetkellä sen verran keskeneräinen, että esimerkkiohjelmia ei saatu käännettyä. Luvun päätteeksi esitetyt esimerkkiohjelmat esittelivät tavoitetilaa, johon Data Parallel Haskellin kehitys pyrkii.

Viimeisessä luvussa esiteltiin tutkielman kokeellinen osuus, jossa tutkittiin miten raskasta laskentaa sisältävän ohjelman muuntaminen rinnakkaislaskentaa hyödyntäväksi onnistuu Glasgow Parallel Haskellin avulla. Koeohjelma laskee satunnaisia desimaalilukuja sisältävien matriisien determinantteja. Koeohjelmasta tehtiin kolme erilaista rinnakkaisversiota ja näiden suoritusaikoja sekä muistin käyttöä verrattiin ohjelman alkuperäiseen perättäisversioon. Kokeen tärkeimpänä havaintona oli, että rinnakkain suoritettavien osien tulee olla riittävän suuria. Pienten ja nopeiden laskutoimitusten suorittamiseksi ei kannata luoda omia säikeitä, vaan ne on nopeampaa suorittaa tavallisena perättäissuorituksena.

Odotukseni olivat erittäin korkealla funktionaalisen ohjelmoinnin suhteen kun ryhdyin sitä tarkemmin tutkimaan. Varsinkin webin keskusteluissa funktionaalinen ohjelmointi mainittiin usein parhaana mahdollisena vaihtoehtona moniydinprosessoreiden hyödyntämiseen tavallisen ohjelmoinnin yhteydessä. Yritin suhtautua varauksella näihin kommentteihin, mutta tästä huolimatta pieni innostus pääsi valtaamaan minut. Onko funktionaalinen ohjelmointi todella näin ongelmatonta kuin näistä keskusteluista saattoi ymmärtää?

Imperatiiviseen ohjelmointiin tottuneelle Haskell ja funktionaalinen ohjelmointi yleisesti vaatii jonkin verran totuttelua. Perinteinen esimerkki quicksortista Haskellilla ja C:llä tehtynä on kiehtova, mutta nopeasti kävi ilmi että vaikka jotkin ongelmat ovat Haskellilla todella yksinkertaisia toteuttaa, ovat toiset taas vaikeampia. Haskell on korkean tason kieli eikä se yritäkään kilpailla C-kielen kanssa, joka on edelleen ykkösvalinta matalan tason ohjelmointiin, vaikka Haskellistakin löytyy työkaluja bittitasolla työskentelyyn.

Haskell-ohjelmoinnin, ja funktionaalisen ohjelmoinnin yleisesti, hieno piirre on se että vaikeasti havaittavia semanttisia virheitä esiintyy ohjelmissa huomattavasti vähemmän kuin

imperatiivisilla kielillä ohjelmoitaessa. Tärkeimpinä tekijöinä tähän ovat Haskellin vahva tyyppijärjestelmä ja funktionaalisen ohjelmoinnin yleinen paradigma, joka korostaa arvojen muutoksia tilan muutosten sijaan. Haskell-ohjelmiin liitetään joskus sanonta: ”if it compiles, it works”, eli jos ohjelma kääntyy niin se toimii. Tämä ei tietenkään aina päde, mutta Haskell-ohjelmointia opetellessa ja testiohjelmia kirjoittaessa tämä havaitsin tämän huomattavan usein todeksi.

Valitettavasti Data Parallel Haskellin kehitys on vielä tutkielman kirjoitushetkellä niin keskeneräinen, että varsinaista koeohjelmaa ei sillä voitu kirjoittaa. Kappaleessa 6 esitellyt esimerkit käyttivät Glasgow Parallel Haskellia rinnakkaisuuden toteuttamiseen. Ensimmäistä rinnakkaisversiota ajaessa tuli selväksi että rinnakkaisuutta voi olla liikaa. Ensimmäisen version suoritusaika oli huomattavasti hitaampi kahdella prosessorilla suoritettuna kuin peräkkäisversion. Tämä johtuu siitä, että uusia tynkiä kipinöitiin lähes kaikissa mahdollisissa kohdissa. Tässä vaiheessa alkoi selkiytyä, että vaikka Glasgow Parallel Haskell huomattavasti helpottaa rinnakkaisuuden hyödyntämistä niin vastuu siitä missä kohdissa rinnakkaisuutta on oikeasti hyödyllistä käyttää jää edelleen ohjelmoijan harteille.

Kuten tuloksista saattoi päätellä, yksittäisen matriisin determinantin laskenta on kannattavinta tehdä peräkkäin ja kipinöidä vain kokonaisten matriisien determinanttien laskeminen. Tilanne olisi kuitenkin toisenlainen, mikäli käytettävissä olisi enemmän prosessoreita kuin aineistossa on matriiseja. Tällöin olisi todennäköisesti kannattavaa kipinöidä lausekkeita myös yksittäisen determinantin laskennan sisällä. Tämä muuttuisi sitä kannattavammaksi mitä suurempia matriiseja aineisto sisältäisi. Esitellyllä aineistolla ja 2–4 prosessorilla kolmas rinnakkaisversio osoittautui suorituskyvyltään parhaaksi. Jos aineisto sisältäisi esimerkiksi 25 kpl 50×50 matriiseja ja käytettävissä olisi kymmeniä tai satoja prosessoreita, saattaisi toinen rinnakkaisversio olla parempi ratkaisu. Yhtä kaikkiin tapauksiin pätevää ratkaisua rinnakkaisuuden toteuttamiseen Glasgow Parallel Haskell ei siis tarjoa, vaan ohjelmoijan on edelleen mietittävä minkälaiseen ympäristöön ja minkälaiselle datalle ohjelma halutaan optimoida, ja erot saattavat olla hyvinkin merkittäviä eri ympäristöissä. Vaikka rinnakkaisuus vaatii edelleen lisätyötä ohjelmoijalta, on rinnakkaisuuden käytännön toteutus kuitenkin helpompaa kuin imperatiivisissa kielissä.

Esimerkkiohjelmien muistin käyttö oli odotettavissa yhtä poikkeusta lukuunottamatta. Pienemmällä aineistoilla matriiseja oli paljon ja kipinöityjä lausekkeita syntyi näin ollen myös paljon. Lausekkeiden kipinöinti ja mahdollinen muuntaminen säikeiksi lisää muistinkäyttöä ja rinnakkaisversiot käyttivätkin pienempien matriisien aineistolla huomattavasti enemmän muistia kuin suuremmalla aineistolla. Odotettavaa olisi että kun kipinöityjen lausekkeiden määrä vähenee niin myös muistin käyttö vähenee. Toisen ja kolmannen rinnakkaisversion tapauksessa näin ei kuitenkaan ollut. Toinen versio, joka synnytti huomattavasti enemmän kipinöityjä lausekkeita kuin kolmas versio, käytti kuitenkin saman verran muistia, jopa aavistuksen vähemmän. Varmaa syytä tähän on vaikea sanoa, mutta mahdollisesti tämä johtui siitä että vaikka toinen versio kipinöi enemmän lausekkeita se kuitenkin loi lopulta vähemmän suoritettavia säikeitä.

Haskellin ja funktionaalisen ohjelmoinnin suurin puute vaikuttaisi olevan suuryritysten tuen puute. Tämähän ei tietenkään ole puute kielessä itsessään. Suuryritysten kuten Microsoftin tai Oraclen kehittämillä ja ylläpitämillä välineillä tehdään ylivoimaisesti suurin osa maailman ohjelmistokehityksestä ja näiden yritysten valitsema kehityssuunta on enemmän tai vähemmän myös yleisen ohjelmistokehityksen suunta. Vaikka Haskellin kehitystä tehdäänkin Microsoftin tutkimusosastolla, ovat yhtiön pääasialliset panostukset imperatiivisten kielien puolella, kuten muillakin suuryrityksillä. Aiemmin rinnakkaisuuteen ei ole prosessoreiden jatkuvan nopeutumisen vuoksi ollut kannattavaa panostaa, mutta nykyisin rinnakkaisuuteen panostetaan huomattavasti enemmän myös imperatiivisten kielten puolella. Esimerkiksi .NET 4:ssä esitellyt Task Parallel Library ja Parallel LINQ tekevät rinnakkaisuuden toteuttamisesta yksinkertaisempaa myös imperatiivisten .NET-kielten puolella, joskaan se ei vieläkään ole yhtä yksinkertaista kuin Glasgow Parallel Haskellissa tai Data Parallel Haskellissa. Kehitys tällä saralla tulee varmasti jatkumaan nopeana imperatiivistenkin kielten kohdalla.

Uskon että kiinnostus Haskelliin ja sen parhaimpiin puoliin tulee kasvamaan jatkossa. Siitä tuskin koskaan tulee ns. mainstream-kieltä, vaikka sillä onkin joitakin kiistattomia etuja imperatiivisiin kieliin nähden. Useimmille ohjelmoijille imperatiivinen paradigma on kuitenkin ohjelmoinnin synonyymi ja tätä asennetta ei helposti muuteta. Haskell on antanut

vaikutteita laajemmin käytetyille kielille, esimerkiksi .NET-ympäristön oman kyselykielen LINQin kehityksen perustana ovat olleet Haskellin monadit. Uskon että Haskellin ja muidenkin funktionaalisten kielten rooli tulee tulevaisuudessa entisestään vahvistumaan uusien ideoiden ja ominaisuuksien kehittäjänä ja vaikutuksien antajana laajan yleisön käyttämille ohjelmointikielille.

LÄHTEET

[Amd67] Amdahl G. M.: Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[Bar11] Barney B.: *Introduction to Parallel Computing*. https://computing.llnl.gov/tutorials/parallel_comp/, December 2011.

[BCH94] Blelloch G., Chatterjee S., Hardwick J. C., Sipelstein J., Zagha M.: Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21:102–111, 1994.

[Ben06] Ben-Ari M.: *Principles of Concurrent and Distributed Programming, 2nd Edition*. Addison Wesley, 2006.

[BlS90] Blelloch G., Sabot G. W.: Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[CLJ07] Chakravarty M. M. T., Leshchinskiy R., Jones S. P., Keller G., Marlow S.: Data parallel Haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM.

[Gus88] Gustafson J. L.: Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[Has12] Haskell Community: *The Haskell 98 Language Report*. http://www.haskell.org/onlinereport/, January 2012.

[HHP07] Hudak P., Hughes J., Peyton Jones S., Wadler P.: A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN confe-*

rence on History of programming languages, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[Hud89] Hudak P.: Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[Hug89] Hughes J.: Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[JMS09] Jones D. J., Marlow S., Singh S.: Parallel Performance Tuning for Haskell. In *Haskell '09: Proceedings of the Second ACM SIGPLAN Symposium on Haskell*. ACM, 2009.

[KCL10] Keller G., Chakravarty M. M., Leshchinskiy R., Peyton Jones S., Lippmeier B.: Regular, shape-polymorphic, parallel arrays in Haskell. *SIGPLAN Not.*, 45(9):261–272, September 2010.

[Lee06] Lee E. A.: *The problem with threads*. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006.

[Lip11] Lipovaca M.: *Learn You a Haskell for Great Good!* No Starch Press, 1st edition, 2011.

[Mei11] Meisel J.: *Task parallelism*. http://cnx.org/content/m15580/1.1/, December 2011.

[MML10] Marlow S., Maier P., Loidl H.-W., Aswad M. K., Trinder P.: Seq no more: Better Strategies for Parallel Haskell. In *Proceedings of the 3rd ACM SIGPLAN symposium on Haskell*, Baltimore, MD, United States, September 2010. ACM Press.

[MPS09] Marlow S., Peyton Jones S., Singh S.: Runtime support for multicore Haskell. *SIGPLAN Not.*, 44:65–78, August 2009.

[OGS08] O'Sullivan B., Goerzen J., Stewart D.: *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[OHL08] Owens J. D., Houston M., Luebke D., Green S., Stone J. E., Phillips J. C.: GPU Computing. In *Proceedings of the IEEE*, volume 96, pages 879–899. IEEE, 2008.

[PCL08] Peyton Jones S., Chakravarty M. M., Leschinskiy R., Keller G.: Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138. Springer-Verlag, 2008.

[Pey02] Peyton Jones S.: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2002.

[Pey12] Peyton Jones S.: *The Haskell Programming Language*. http://www.haskell.org/haskellwiki/Haskell, January 2012.

[PJW93] Peyton Jones S. L., Wadler P.: Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

[Roj97] Rojas R.: *A Tutorial Introduction to the Lambda Calculus*. FU Berlin, 1997.

[THL96] Trinder P. W., Hammond K., Loidl H.-W., Peyton Jones S., Wu J.: Accidents always Come in Threes: A Case Study of Data-intensive Programs in Parallel Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1996.

[THL98] Trinder P. W., Hammond K., Loidl H.-W., Peyton Jones S.: Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

Liite A

DETERMINANTS

Ohjelma lukee matriisit tekstitiedostosta ja tulostaa tiedostoon niiden determinantit

```
module Main where


import System.Environment (getArgs)


type Matrix = [[Double]]
type Point = (Int, Int)


-- Main lukee matriisit tiedostosta ja kirjoittaa niiden determinantit toiseen tiedostoon.
main :: IO ()
main = do
  args ← getArgs
  contents ← readFile (args !! 0)
  let matrices = map (\s → read s :: Matrix) (lines contents)
  writeFile (args !! 1) (show (map determinant matrices))


-- Determinantin laskennan käynnistys
determinant :: Matrix → Double
determinant m = calcDet (length m) (doubleM m)


-- Rekursiivinen determinantin laskenta Laplace - kehitelmällä.
-- Matriisin determinantti voidaan laskea sen alideterminanttien avulla
-- Esim. matriisin 1 2 3 determinantti on
                -- 4 5 6
                -- 7 8 9
-- on 1 * det(5 6 - 2*det(4 6 +3*det(4 5
```

```
--            8 9)        7 9)       7 8)
calcDet :: Int → Matrix → Double
-- Yhden elementin matriisin determinantti on luku itse
calcDet 1 mat = head (head mat)
calcDet size mat =
        sum (zipWith (*) coeffs (map (\m → calcDet (size-1) m) (map doubleM minors)))
               where
                -- Ensimmäisen rivin alkiot kertoimiksi
                coeffs = zipWith (*) (repList size [1, isNeg (size-1)]) (head mat)

                -- Alimatriisit
                minors = [minor (2,i) (size,i+(size-2)) mat | i ← [2..(size + 1)]]

-- Apufunktio, joka tekee duplikaatin jokaisesta rivistä
-- alimatriisien laskennan helpottamiseksi.
doubleM :: Matrix → Matrix
doubleM [] = []
doubleM (x:xs) = (x ++ x) :  doubleM xs

-- Alimatriisien determinanttien kertoimien merkki.
isNeg n = if odd n then -1 else 1

-- Yhden alimatriisin laskenta. Pisteparametrit määrittävät
-- mistä alkiosta mihin alkioon alimatriisi haetaan
-- Esim. minor (2,2) (3,3) 1 2 3
--                         4 5 6
--                         7 8 9
-- hakee matriisin 5 6
--                 8 9
minor :: Point → Point → Matrix → Matrix
minor (x1,y1) (x2,y2) m =  column2
                    where
```

```
    row1 = drop (x1 - 1) m
    column1 = map (drop (y1-1)) row1
    row2 = take (x2-x1 + 1) column1
    column2 = map (take (y2-y1 + 1)) row2


-- Kertoimien merkin laskennan apufunktio.
repList :: Int → [a] → [a]
repList 0 _ = []
repList n xs = xs ++ repList (n-1) xs
```

Liite B

DETERMINANTS, ENSIMMÄINEN RINNAKKAISVERSIO

```
module Main where

import System.Environment (getArgs)
import Control.Parallel.Strategies

type Matrix = [[Double]]
type Point = (Int, Int)

-- Main lukee matriisit tiedostosta ja tulostaa niiden determinantit toiseen tiedostoon.
main :: IO ()
main = do
  args ← getArgs
  contents ← readFile (args !! 0)
  let matrices = parMap rseq (\s → read s :: Matrix) (lines contents)
  writeFile (args !! 1) (show (parMap rdeepseq determinant matrices))

-- Determinantin laskennan käynnistys
determinant :: Matrix → Double
determinant m = calcDet (length m) (doubleM m)

-- Rekursiivinen determinantin laskenta Laplace - kehitelmällä.
-- Matriisin determinantti voidaan laskea sen alideterminanttien avulla
-- Esim. matriisin 1 2 3 determinantti on 1 * det(5 6 - 2*det(4 6 +3*det(4 5
                -- 4 5 6                          8 9)        7 9)       7 8)
                -- 7 8 9
```

```
calcDet :: Int → Matrix → Double
-- Yhden elementin matriisin determinantti on luku itse
calcDet 1 mat = head (head mat)
calcDet size mat =
 sum (zipWith (*) coeffs (parMap rseq (\m → calcDet (size-1) m) (parMap rseq doubleM minors)
  where
   -- Ensimmäisen rivin alkiot kertoimiksi
   coeffs = zipWith (*) (repList size [1, isNeg (size-1)]) (head mat) `using` parList rseq
   -- Alimatriisit
   minors = [minor (2,i) (size,i+(size-2)) mat | i ← [2..(size + 1)]] `using` parList rseq

-- Apufunktio, joka tekee duplikaatin jokaisesta rivistä.
-- Tällä helpotetaan alimatriisien laskentaa.
doubleM :: Matrix → Matrix
doubleM [] = []
doubleM (x:xs) = runEval $ do
                          nexts ← rpar (x ++ x)
                          rest ← rseq (doubleM xs)
                          return (nexts : rest)

-- Alimatriisien determinanttien kertoimien merkki.
isNeg n = if odd n then -1 else 1

-- Yhden alimatriisin laskenta.
-- Pisteparametrit määrittävät mistä alkiosta mihin alkioon alimatriisi haetaan
-- Esim. minor (2,2) (3,3) 1 2 3
--                         4 5 6
--                         7 8 9
-- hakee matriisin 5 6
--                 8 9
minor :: Point → Point → Matrix → Matrix
minor (x1,y1) (x2,y2) m =  column2
```

```
                    where
                      row1 = drop (x1 - 1) m
                      column1 = parMap rdeepseq (drop (y1-1)) row1
                      row2 = take (x2-x1 + 1) column1
                      column2 = parMap rdeepseq (take (y2-y1 + 1)) row2


-- Kertoimien merkin laskennan apufunktio.
repList :: Int → [a] → [a]
repList 0 _ = []
repList n xs = runEval $ do rest ← rpar (repList (n-1) xs)
                            return (xs ++ rest)
```

Liite C

DETERMINANTS, TOINEN RINNAKKAISVERSIO

```
module Main where

import System.Environment (getArgs)
import Control.Parallel.Strategies

type Matrix = [[Double]]
type Point = (Int, Int)

-- Main lukee matriisit tiedostosta ja tulostaa niiden determinantit toiseen tiedostoon.
main :: IO ()
main = do
  args ← getArgs
  contents ← readFile (args !! 0)
  let matrices = map (\s → read s :: Matrix) (lines contents)
  writeFile (args !! 1) (show (parMap rdeepseq determinant matrices))

-- Determinantin laskennan käynnistys
determinant :: Matrix → Double
determinant m = calcDet (length m) (doubleM m)

-- Rekursiivinen determinantin laskenta Laplace - kehitelmällä.
-- Matriisin determinantti voidaan laskea sen alideterminanttien avulla
-- Esim. matriisin 1 2 3 determinantti on 1 * det(5 6 - 2*det(4 6 +3*det(4 5
                -- 4 5 6                          8 9)        7 9)       7 8)
                -- 7 8 9
calcDet :: Int → Matrix → Double
-- Yhden elementin matriisin determinantti on luku itse
```

```
calcDet 1 mat = head (head mat)
calcDet size mat =
 sum (zipWith (*) coeffs (parMap rseq (\m → calcDet (size-1) m) (parMap rseq doubleM minors))
  where
   -- Ensimmäisen rivin alkiot kertoimiksi
   coeffs = zipWith (*) (repList size [1, isNeg (size-1)]) (head mat) `using` parList rseq
   -- Alimatriisit
   minors = [minor (2,i) (size,i+(size-2)) mat | i ← [2..(size + 1)]] `using` parList rseq

-- Apufunktio, joka tekee duplikaatin jokaisesta rivistä.
-- Tällä helpotetaan alimatriisien laskentaa.
doubleM :: Matrix → Matrix
doubleM [] = []
doubleM (x:xs) = (x ++ x) :  doubleM xs

-- Alimatriisien determinanttien kertoimien merkki.
isNeg n = if odd n then -1 else 1

-- Yhden alimatriisin laskenta. Pisteparametrit määrittävät mistä alkiosta
-- mihin alkioon alimatriisi haetaan
-- Esim. minor (2,2) (3,3) 1 2 3
--                         4 5 6
--                         7 8 9
-- hakee matriisin 5 6
--                 8 9
minor :: Point → Point → Matrix → Matrix
minor (x1,y1) (x2,y2) m =  column2
                      where
                        row1 = drop (x1 - 1) m
                        column1 = map (drop (y1-1)) row1
                        row2 = take (x2-x1 + 1) column1
                        column2 = map (take (y2-y1 + 1)) row2
```

```
-- Kertoimien merkin laskennan apufunktio.
repList :: Int → [a] → [a]
repList 0 _ = []
repList n xs = xs ++ repList (n-1) xs
```

Liite D

DETERMINANTS, KOLMAS RINNAKKAISVERSIO

```
module Main where

import System.Environment (getArgs)
import Control.Parallel.Strategies

type Matrix = [[Double]]
type Point = (Int, Int)

-- Main lukee matriisit tiedostosta ja tulostaa niiden determinantit toiseen tiedostoon.
main :: IO ()
main = do
  args ← getArgs
  contents ← readFile (args !! 0)
  let matrices = map (\s → read s :: Matrix) (lines contents)
  writeFile (args !! 1) (show (parMap rdeepseq determinant matrices))

-- Determinantin laskennan käynnistys
determinant :: Matrix → Double
determinant m = calcDet (length m) (doubleM m)

-- Rekursiivinen determinantin laskenta Laplace - kehitelmällä.
-- Matriisin determinantti voidaan laskea sen alideterminanttien avulla
-- Esim. matriisin 1 2 3 determinantti on 1 * det(5 6 - 2*det(4 6 +3*det(4 5
               -- 4 5 6                          8 9)        7 9)       7 8)
               -- 7 8 9
calcDet :: Int → Matrix → Double
-- Yhden elementin matriisin determinantti on luku itse
```

```
calcDet 1 mat = head (head mat)
calcDet size mat =
 sum (zipWith (*) coeffs (map (\m → calcDet (size-1) m) (map doubleM minors)))
  where
   -- Ensimmäisen rivin alkiot kertoimiksi
   coeffs = zipWith (*) (repList size [1, isNeg (size-1)]) (head mat)
   -- Alimatriisit
   minors = [minor (2,i) (size,i+(size-2)) mat | i ← [2..(size + 1)]]

-- Apufunktio, joka tekee duplikaatin jokaisesta rivistä.
-- Tällä helpotetaan alimatriisien laskentaa.
doubleM :: Matrix → Matrix
doubleM [] = []
doubleM (x:xs) = (x ++ x) :  doubleM xs

-- Alimatriisien determinanttien kertoimien merkki.
isNeg n = if odd n then -1 else 1

-- Yhden alimatriisin laskenta. Pisteparametrit määrittävät mistä alkiosta
-- mihin alkioon alimatriisi haetaan
-- Esim. minor (2,2) (3,3) 1 2 3
--                         4 5 6
--                         7 8 9
-- hakee matriisin 5 6
--                 8 9
minor :: Point → Point → Matrix → Matrix
minor (x1,y1) (x2,y2) m =  column2
                      where
                        row1 = drop (x1 - 1) m
                        column1 = map (drop (y1-1)) row1
                        row2 = take (x2-x1 + 1) column1
                        column2 = map (take (y2-y1 + 1)) row2
```

```
-- Kertoimien merkin laskennan apufunktio.
repList :: Int → [a] → [a]
repList 0 _ = []
repList n xs = xs ++ repList (n-1) xs
```