Simo Rinne

Suorituskyvyn parantaminen reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä

Tietotekniikan pro gradu -tutkielma

3. marraskuuta 2017

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Simo Rinne

Yhteystiedot: rinne.simo@gmail.com

Ohjaajat: Ville Tirronen ja Antti-Jussi Lakanen

Työn nimi: Suorituskyvyn parantaminen reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä

Title in English: Improving performance in games made with functional reactive programming

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Pelit ja pelillisyys

Sivumäärä: 73+0

Tiivistelmä: Tämän pro gradu -tutkielman tavoitteena on tutkia, miten suorituskykyä voi parantaa reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä. Tutkielmassa tuotettiin suunnittelutieteen menetelmien mukaisesti IT-artefakti, jolla pystyy rinnakkaistamaan peliobjektien päivityksen reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä. Suorituskykymittausten perusteella IT-artefakti paransi mittauksessa käytetyn testipelin suorituskykyä.

Avainsanat: reaktiivinen ohjelmointi, funktio-ohjelmointi, peliohjelmointi, suorituskyky

Abstract: The purpose of this master's thesis is to study how performance can be improved in games made with reactive functional programming. Design science method was used to create an IT artifact that improves performance in games made with reactive functional programming by updating game objects in parallel. The performance tests conducted in this study show that applying the IT artifact to a test game resulted in increased performance.

Keywords: reactive programming, functional programming, game programming, performance

Termiluettelo

FRP	Reaktiivinen funktio-ohjelmointi. Ohjelmointiparadigma, jossa kirjoitetaan ajan suhteen muuttuvia funktioita, eli signaaleja.
Funktio-ohjelmointi	Deklaratiivinen ohjelmointiparadigma, jossa funktiot ovat tärkeimmässä asemassa.
GHC	Glasgow Haskell Compiler. Haskell-ohjelmointikielen kääntäjä.
Haskell	Yleiskäyttöinen staattisesti tyypitetty puhtaasti funktionaalinen ohjelmointikieli, jolla on akateemiset juuret.
HEC	Haskell Execution Context. Jokaista käytössä olevaa käyttöjärjestelmän säiettä kohden luotava laskentakonteksti.
Normal form	Loppuun asti laskettu lauseke.
Reaktiivinen ohjelmointi	Ohjelmointiparadigma, jossa määritellään arvoja, jotka ovat jatkuvassa suhteessa toisiin arvoihin.
RTS	GHC runtime system. Ajoympäristö, joka linkitetään mukaan Haskell-ohjelmiin.
Weak head normal form	Pienimmän mahdollisimman määrän verran laskettu lauseke.

Kuviot

Taulukot

Listaukset

Sisältö

1 Johdanto

Yhä useampi viettää enemmän ja enemmän aikaa ruudun äärellä. Sosiaalisen median ja muiden viihdykkeiden rinnalla pelit ja pelaaminen kasvattavat jatkuvasti suosiotaan. Suomen pelialan yhdistyksen Neogames (2016) mukaan vuonna 2016 maailmassa julkaistiin arviolta noin 760 000 peliä. Mittava tarjonta tekee pelialasta haasteellisen toimijoilleen, jolloin myös pelinkehittämisen tutkimus erilaisista näkökulmista on tärkeää.

Tässä tutkimuksessa tarkastellaan reaktiivista funktio-ohjelmointia pelinkehityksen näkökulmasta. Vaikka funktio-ohjelmointi on lähtöisin akateemisista piireistä, sitä hyödynnetään nykyään myös ohjelmistoteollisuudessa. Ericssonin alunperin kehittämää funktionaalista Erlang-ohjelmointikieltä käytetään kehittämään hajautettuja ja virheensietokykyisiä järjestelmiä (Armstrong 2007). Esimerkiksi Facebook käyttää Erlang-, ML-, ja Haskell-ohjelmointikieltä (Piro ja Letuchy 2009) ja Keera Studios on käyttänyt Haskell-ohjelmointikieltä ja reaktiivista funktio-ohjelmointia kaupallisten Android ja iOS mobiilipelien kehittämiseen (Maruseac 2017; Perez, Bärenz ja Nilsson 2016).

Funktio-ohjelmoinnin käyttö on viime vuosien aikana yleistynyt. Suosio näkyy erityisesti palvelinsovellusten ohjelmointikielten valinnoissa ja yleisimpien imperatiivisten ohjelmointikielten ominaisuuksissa. Esimerkiksi Java SE 8:aan on lisätty funktio-ohjelmointia lambda-lausekkeiden muodossa (Kyyppö ja Vesterholm 2015). Funktio-ohjelmoinnin käytön yleistymisen myötä reaktiivisen funktio-ohjelmoinnin tarkasteleminen pelinkehittämisen näkökulmasta on tuore ja ajankohtainen tutkimusaihe. Aihetta on aikaisemmin tutkittu vain vähän, eikä aiemmissa tutkimuksissa ole otettu kantaa suorituskykyyn.

Reaktiivinen funktio-ohjelmointi soveltuu hyvin peliohjelmointiin (Perez, Bärenz ja Nilsson 2016; Courtney, Nilsson ja Peterson 2003). Funktio-ohjelmoinnissa suorituskyvyn arviointi on kuitenkin haastavaa ja suorituskyky voi tietyissä tapauksissa olla heikko (Sutter ja Larus 2005). Tämän tutkimuksen tarkoituksena on esittää kei-

no parantaa suorituskykyä reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä rinnakkaistamalla pelilogiikan laskentaa usealle prosessorin ytimelle.

Tutkimuksessa selvitetään, miten suorituskykyä voi parantaa reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä. Tutkimusongelmaa lähdettiin ratkaisemaan suunnittelutieteen periaatteiden mukaisesti ja tutkimuksessa toteutettiin IT-artefaktina funktio, jonka avulla Netwire-kirjaston signaalifunktiokokoelmia voi evaluoida rinnakkaisesti. Funktio on yleiskäyttöinen ja sen voi ottaa käyttöön myös aikaisemmin Netwire-kirjastolla tehtyihin peleihin tai muihin sovelluksiin, joissa käsitellään signaalifunktiokokoelmia.

Tässä tutkimuksessa keskeisiä käsitteitä ovat funktio-ohjelmointi ja reaktiivinen ohjelmointi. Funktio-ohjelmointi lasketaan deklaratiiviseksi ohjelmoinniksi, jossa ohjelman logiikka kuvataan ilmaisematta eksplisiittisesti suoritusjärjestystä. Reaktiivisessa ohjelmoinnissa määritellään arvoja, jotka ovat jossain suhteessa toisiin arvoihin. Alkuperäisten arvojen muuttuessa niistä riippuvat arvot päivittyvät myöskin. Reaktiivinen ohjelmointi vastaa olio-ohjelmointikielissä hyvin pitkälti tarkkailijamallia (engl. *observer pattern*). Reaktiivista ohjelmointia soveltaessa funktionaaliseen ohjelmointiin puhutaan reaktiivisesta funktio-ohjelmoinnista (engl. *functional reactive programming*, FRP), jossa kirjoitetaan ajan suhteen muuttuvia funktioita, eli signaaleja.

Tutkielma jakautuu teoriaosuuteen, tutkimusosioon ja tuloksiin. Teoriaosuudessa rakennetaan pohjaa tutkimusosiolle ja sen toisena tavoitteena on tarjota lukijalle perustiedot tässä tutkimuksessa toteutetun IT-artefaktin ymmärtämiseen. Teoriaosuus koostuu reaktiivisesta ohjelmoinnista, funktio-ohjelmoinnista, reaktiivisesta funktio-ohjelmoinnista ja rinnakkaislaskennasta. Teoriaosuuden lopussa esitellään tapoja parantaa suorituskykyä reaktiivisessa funktio-ohjelmoinnissa pohjautuen aikaisempaan tutkimukseen. Luvussa 3 pureudutaan suorituskyvyn parantamiseen esitellen alussa tarkemmin tutkimusongelma ja -metodi, sekä kuvaillaan tutkimuksen kulku. Tutkimusosio etenee rinnakkaistamisen toteuttamisen ja testiympäristön esittelemisen kautta mittausmenetelmiin ja mittaukseen vaikuttaviin parametreihin. Luvussa 4 esitellään suoritetun suorituskykymittauksen tulokset. Viidennen luvun pohdinnoissa käsitellään tulosten lisäksi niiden merkitystä ja luotettavuutta sekä

arvioidaan miten tutkimusongelman ratkaisu onnistui. Kuudes luku on yhteeveto, jossa palataan tutkimuksen tutkimusongelmaan, kootaan yhteen päätulokset ja esitetään jatkotutkimusaiheet.

2 Teoreettinen viitekehys

Tässä luvussa esitellään tutkimuksen teoreettinen viitekehys. Pelejä kehitettäessä reaktiivisella funktio-ohjelmoinnilla peliobjekteja kuvataan signaalifunktioilla. Peleissä objekteja on usein paljon, jolloin objekteista muodostuvaa signaalifunktiokokoelmaa on usein tarve hallinnoida. Signaalifunktiokokoelman jokaiselle signaalille pitää ohjata syötettä ja laskea signaalifunktion tuottama arvo. Jotkin funktio-ohjelmointikirjastot, esimerkiksi Yampa, toteuttavat tähän tarvittavat toiminnallisuudet valmiiksi, mutta tutkimukseen valittu Netwire-kirjasto ei toteuta. Netwire-kirjasto valittiin tutkimukseen, koska Yampa-kirjaston signaalifunktiokokoelman laskentafunktioiden rinnakkaistaminen on ongelma, jota Yampa-kirjaston kehittäjät eivät ole saaneet ratkaistua (Perez 2017). Netwire-kirjaston signaalifunktiokokoelman arvon laskentaan löytyy mallitoteutus luvusta 2.3.5. Yampa- ja Netwire-kirjastoista kerrotaan tarkemmin luvussa 2.3.1.

Tämän luvun alussa käsitellään lyhyesti reaktiivista ohjelmointia ja funktio-ohjelmointia. Nämä kappaleet tukevat reaktiivisen funktio-ohjelmoinnin osuutta, joka päättyy tutkimuksen kannalta olennaiseen signaalifunktioiden evaluaatioon. Sen jälkeen tarkastellaan rinnakkaislaskentaa, sekä siihen liittyvää Ahmdalin lakia, johon tutkimuksen tuloksia tullaan peilaamaan. Viimeisenä tutustutaan aikaisempaan tutkimukseen liittyen suorituskyvyn parantamiseen reaktiivisessa funktio-ohjelmoinnissa.

2.1 Reaktiivinen ohjelmointi

Ensimmäiset viittaukset reaktiivisuuteen löytyvät jo 80-luvulta. Harel ja Pnueli (1985) ovat määritelleet reaktiivisen systeemin olevan jatkuvassa interaktiossa ympäristönsä kanssa. Systeemi aktivoituu saamastaan syötteestä ja tuottaa sille vasteen.

Reaktiivinen ohjelmointi on ohjelmointiparadigma, joka keskittyy ajan suhteen muuttuviin arvoihin ja muutoksen eteenpäin välittämiseen (Bainomugisha ym. 2013). Reaktiivista ohjelmointia voi käyttää esimerkiksi käyttöliittymissä, peleissä tai missä tahansa ohjelmissa, joissa käyttäjä on jatkuvassa interaktiossa ohjelman kanssa

(Boussinot 1991; Harel ja Pnueli 1985). Ohjelman interaktiivisin osuus on usein käyttöliittymä, josta lähetetään jatkuvasti tapahtumia, joihin ohjelman tulee reagoida (Bainomugisha ym. 2013). Tälläisten ohjelmien teko on haasteellista perinteisin menetelmin ja sen takia ne tehdään käyttäen asynkronisia tapahtumankäsittelijöitä tai takaisinkutsuja (engl. *callback*) (Bainomugisha ym. 2013).

Reaktiivisen ohjelmoinnin käyttö ei rajaa toteutettavassa ohjelmassa ohjelmointikielen valintaa. Reaktiivista ohjelmointia voi tehdä imperatiivisilla ohjelmointikielillä, kuten esimerkiksi C-kielellä (Boussinot 1991), Javalla, Pythonilla ja C#:lla (Bainomugisha ym. 2013) tai deklaratiivisilla kielillä, kuten esimerkiksi Haskelilla (Bainomugisha ym. 2013) ja Standard ML:llä (Pucella 1998), tai millä tahansa ohjelmointikielellä, jossa käytetään call-by-value-parametrinvälitysmekanismia (Cooper 2008).

2.1.1 Reaktiivisen ohjelmoinnin toimintaperiaate

Listauksen 1 esimerkissä määritetään kaksi numeerista muuttujaa ja kolmanteen lasketaan niiden summa. Imperatiivisessa ohjelmoinnissa `var3` muuttuja sisältää aina arvon 3, vaikka `var1` ja `var2` muuttujien arvot muuttuisivat. Reaktiivisessa ohjelmoinnissa `var3` muuttuja pidetään aina ajan tasalla laskemalla sen arvo automaattisesti uudestaan aina kun muuttuja `var1` tai `var2` muuttuu (Bainomugisha ym. 2013).

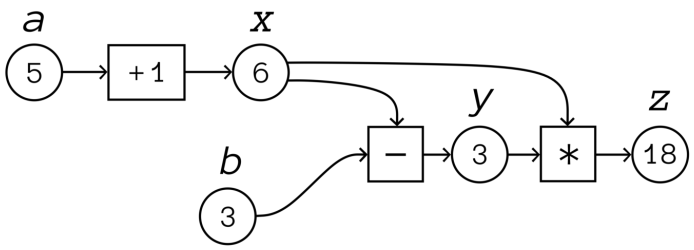
Listaus 1. Kahden luvun summaaminen reaktiivisesti (Bainomugisha ym. 2013).

```
var1 = 1
var2 = 2
var3 = var1 + var2
```

Muuttujista muodostuu reaktiivisessa ohjelmoinnissa riippuvuuksien verkosto, jossa lähtöarvojen muuttaminen päivittyy automaattisesti koko verkon läpi. Listauksen 2 koodista muodostuvaa riippuvuuksien verkostoa on havainnollistettu kuviossa 1. Muuttujan z arvo riippuu x ja y muuttujista, jotka ovat riippuvaisia vakioluvuista tai alkuarvoista. Alkuarvojen a ja b muuttuessa muuttujan z arvo lasketaan uudestaan verkoston mukaisesti.

Listaus 2. Monimutkaisempi esimerkki reaktiivisesta laskennasta.

```
a = 5
b = 3
x = a + 1
y = x - b
z = x * y
```



Kuvio 1. Esimerkki reaktiivisten arvojen riippuvuuksien verkosta.

Ohjelmoijan näkökulmasta arvojen päivittyminen tapahtuu automaattisesti. Ohjelmointikielen tasolla reaktiivista kieltä tai kirjastoa suunnitellessa valitaan toteutustavaksi vetopohjaisuus tai työntöpohjaisuus. Työntöpohjaisessa lähestymistavassa heti kun uutta tietoa tulee saataville, esimerkiksi tapahtuman tullessa, tiedonlähteet työntävät tietoa riippuvuusverkon läpi niitä tarvitseville kuluttajille. Vetopohjaisessa lähestymistavassa arvoa päivittäessä tietoa vedetään lähteistä, joista arvo riippuu. (Bainomugisha ym. 2013.)

2.2 Funktio-ohjelmointi

Tässä tutkielmassa toteutettu IT-artefakti ja mittausympäristö ovat toteutettu Haskell-ohjelmointikielellä. Tässä luvussa esitellään funktio-ohjelmointia ja Haskell-ohjelmointikielen syntaksia, jotta tutkielman myöhempiä osia pystyy ymmärtämään. Haskell-ohjelmointikieli on yleiskäyttöinen puhtaasti funktionaalinen ohjelmointikieli, jolla on akateemiset juuret. Haskell-ohjelmointikieli tarjoaa ohjelmoijalle muun muassa korkeamman asteen funktiot, staattisen polymorfismin, algebralliset tietotyypit, muodonsovituksen, monipuoliset primitiivitietotyypit ja laiskan laskennan

(Hudak ym. 2003). Tämän tutkielman listauksissa on käytetty notaatiota, jonka vastaavuudet Haskell lähdekoodiin on esitetty taulukossa 1 selkeyden varmistamiseksi.

Taulukko 1. Listauksissa käytetyn notaation vastaavuudet lähdekoodiin

Notaatio	Vastaavuus lähdekoodissa
$\rightarrow$	`->`
$\Rightarrow$	`=>`
$\prec$	`-<`

2.2.1 Tyypit ja arvot

Haskell-ohjelmointikieli on staattisesti tyypitetty ohjelmointikieli. Staattinen tyypitys varmistaa, että ohjelmoija ei ole vahingossa käyttänyt väärän tyyppisiä arvoja väärässä paikassa. Yksi staattisen tyypityksen eduista on se, että tyyppivirheet havaitaan ohjelman käännöksen aikana ja se antaa kääntäjälle mahdollisuuden optimoida ohjelmaa paremmin. (Hudak ja Fasel 1992.)

Haskell-ohjelmointikielessä kaikki laskenta tapahtuu laskemalla lausekkeille arvoja (Hudak ja Fasel 1992). Jokaisella arvolla on tyyppi ja koska lausekkeet kuvastavat laskematonta arvoa, jokaisella lausekkeella on myös tyyppi. Listauksessa 3 on esitetty esimerkkejä arvoista ja niiden tyypeistä. Listauksessa oleva `::` voidaan lukea "on tyypiltään", esimerkiksi listauksessa oleva arvo `(True, 'a')`, joka kuvastaa arvoparia (engl. *tuple*) on tyypiltään `(Bool, Char)`.

Listaus 3. Esimerkkejä arvojen tyypeistä.

```
5.8          :: Double
(True, 'a')  :: (Bool, Char)
[1, 2, 3, 4] :: [Int]
funktio      :: Double → Double
```

Listat ovat tyypiltään polymorfisia ja rakenteeltaan rekursiivisia (Hudak ja Fasel 1992). Lista voi sisältää vain samantyyppisiä arvoja. Listan polymorfisuus tarkoittaa sitä, että listan tyyppi `[a]`, kuvastaa tyyppiperhettä, johon sisältyy jokaista tyyppiä `a` kohden tyyppi, joka on lista `a`:n tyypeistä. Kaikki tyyppien nimet kirjoitetaan isolla

alkukirjaimella ja tyyppimuuttujat pienellä alkukirjaimella. Edellisessä esimerkissä `a` on tyyppimuuttuja ja listauksessa 3 olevat `Double`, `Char` ja `Int` ovat tyyppejä.

Haskell-ohjelmointikielestä löytyy `Either`-tietotyyppi, joka on määritelty listauksessa 4 esitellyllä tavalla. `Either` on listan tapaan polymorfinen tyyppi, mutta sillä on kaksi tyyppimuuttujaa yhden sijaan. `Left` ja `Right` ovat konstruktoreita. Esimerkiksi tyypin `Either String [Int]` arvoja voisi olla `Left "kissa"` ja `Right [5, 7]`. Listauksessa 4 käytetään avainsanaa *`data`*, jonka avulla voidaan määrittää uusia tyyppejä. Määritettäessä uutta tyyppiä *`data`*-sanaa seuraa tyypin nimi (esimerkissä `Either`), jonka jälkeen annetaan parametrit (esimerkissä `a` ja `b`). Määrittely päätetään yhtä suuri kuin -merkkiin ja konstruktoreihin (esimerkissä `Left` ja `Right`). (Hudak ja Fasel 1992.)

Listaus 4. Either-tietotyyppi (Hudak ym. 2003).

```
data Either a b = Left a | Right b
```

Tyyppisynonyymejä voi määrittää `type`-avainsanalla (Hudak ja Fasel 1992). Tyyppisynonyymi määrittää jo olemassa olevalle tyypille uuden nimen. Uudet nimet ovat usein alkuperäisiä nimiä lyhyempiä ja kuvaavat paremmin alkuperäistä tyyppiä. Tällä pyritään parantamaan koodin luettavuutta. Esimerkki tyyppisynonyymin määrittelystä löytyy listauksesta 17 (s. 19), jossa määritetään `WireP s e`-tyyppi, joka on `Wire s e Identity`-tyypin tyyppisynonyymi.

2.2.2 Funktiot ja tyyppiluokat

Haskell-ohjelmointikielessä funktioita voi määrittää listauksessa 5 esitetyllä tavalla. Ensimmäinen rivi on funktion tyypimäärittely ja toisella rivillä on funktion toteutus. Funktio `keskiarvo` ottaa parametrina kaksi desimaalilukua (`Float`) ja palauttaa desimaaliluvun. Funktion tyypissä nuoli on oikealle assosiatiivinen, eli sen voi ajatella olevan `Float → (Float → Float)`, eli funktio, joka ottaa desimaaliluvun ja palauttaa funktion, joka on tyypiltään `Float → Float`.

Listaus 5. Funktioiden määrittely.

```
keskiarvo :: Float → Float → Float
keskiarvo x y = (x + y) / 2
```

Funktioita pystyy kutsumaan eri tavoin. Kutsu voidaan muotoilla kirjoittamalla ensin funktion nimi ja sen jälkeen parametrit (Hudak ja Fasel 1992). Lausekkeen `keskiarvo 3.6 1.5` arvo on `2.55`. Parametrit eivät tule sulkuihin kuten monissa muissa ohjelmointikielissä. Osittain kutsuttaessa (engl. *partial application*) annetaan vain osa parametreistä (Hudak ja Fasel 1992). Esimerkiksi lausekkeen `keskiarvo 3.6` arvo olisi funktio, joka ottaa vielä toisen parametrin ja palauttaa desimaaliluvun. Funktioita voi kutsua myös *infix* muodossa laittamalla funktio parametrien väliin (Hudak ja Fasel 1992). Esimerkiksi funktiokutsu lausekkessa `3.6 ‘keskiarvo‘ 1.5` on sama kuin `keskiarvo 3.6 1.5`.

Haskell-kielessä, kuten muissakin ohjelmointikielissä, voi tehdä anonyymejä funktioita (Hudak ja Fasel 1992). Esimerkiksi listauksessa 5 oleva `keskiarvo`-funktio on identtinen funktion `\x → \y →(x + y) / 2` kanssa, joka on lyhyemmin kirjoitettuna `\x y →(x + y) / 2`.

Haskell tukee Hindley-Milner-tyylisen parametrisen polymorfismin lisäksi ad-hoc polymorfismia tyyppiluokilla (Hudak ym. 2003). Ad-hoc polymorfismi on toiselta nimeltään kuormitus (Hudak ja Fasel 1992). Tällä saavutetaan se, että sama funktio pystyy toimimaan useilla eri tietotyypeillä, esimerkiksi summafunktio `(+)` pystyy lisämään yhteen kokonaislukujen lisäksi myös desimaalilukuja.

Tyyppiluokat koostuvat tyyppimääritelmistä ja niitä voi määritellä listauksessa 6 esitetyllä tavalla. Listauksessa on määritetty `Monoid`-tyyppiluokka, jossa on kaksi tyyppimääritelmää. `mempty` kuvaa tyhjää arvoa ja `mappend` kuvaa tavan yhdistää arvoja siten, että saadaan uusi samantyyppinen alkio. Tyyppiluokan toteuttavan tyypin täytyy tehdä toteutukset kaikille tyyppiluokan määritelmille. Toteutukset tehdään tyyppiluokkainstanssiin. Listauksessa 7 on esimerkki `Monoid`-tyyppiluokan toteuttamisesta `String`-tyypille. Listauksen esimerkissä `mempty` on tyhjä merkkijono ja `mappend` on funktio, joka lisää Haskell-kielestä valmiina löytyvällä `(++)`-funktiolla

merkkijonoja yhteen.

Listaus 6. `Monoid`-tyyppiluokan määritelmä (Hackage 2017b).

```
class Monoid a where
    mempty  :: a
    mappend :: a → a → a
```

Listaus 7. Esimerkki `Monoid`-tyyppiluokan instanssista.

```
instance Monoid String where
    mempty      = ""
    mappend a b = a ++ b
```

Funktioihin voi laittaa tyyppiluokkarajoitteita (Hudak ja Fasel 1992). Sen sijaan, että funktio ottaa jonkin tietyn tyypin, se voi ottaa minkä tahansa tyypin, joka toteuttaa tietyn tyyppiluokan määritelmät. Tyyppiluokkarajoitteet merkataan funktioon ennen nuolimerkintää (⇒). Listauksessa 8 on `summa`-funktio, joka laskee rekursiolla yhteen kaikki parametrina tulleen listan alkiot `mappend`-funktion avulla. Yhteenlisäämisen toteutus riippuu tapauskohtaisesti tyypistä ja sen `Monoid`-tyyppiluokan toteutuksesta. Funktiossa voi olla myös useampia tyyppiluokkarajoitteita.

Listaus 8. Esimerkki tyyppiluokkarajoitteesta.

```
summa :: Monoid a ⇒ [a] → a
summa []             = mempty
summa (alkio : loput) = alkio `mappend` summa loput
```

2.2.3 Monadit ja nuolet

Haskell-ohjelmointikielessä sivuvaikutuksia ja imperatiivista ohjelmointia voi mallintaa monadeilla (Wadler 1997). Tässä luvussa on käsitellään tutkimuksen kannalta olennaisia monadeja vain pintapuolisesti, sillä monadit ovat haastava konsepti ymmärtää (Hudak ja Fasel 1992). Monadin käsitettä avataan esimerkin avulla yleisellä tasolla ja esitellään syntaksia sen verran, että tutkimuksen muita osia on helpompi ymmärtää. Monadien jälkeen luvussa käsitellään nuolia, joilla on samoja piirteitä kuin monadeilla. Nuolilla kuvataan laskentaa abstraktilla tasolla monipuolisemmin

kuin monadeilla.

Monadit pohjautuvat kategoriateorian käsitteisiin. Monadien toiminnan ymmärtäminen ei kuitenkaan vaadi kategoriateorian ymmärtämistä. Monadit ovat tärkeitä ja usein käytettyjä työkaluja, jotka löytyvät peruskirjastoista valmiina listauksessa 9 esitetyn Monad-tyyppiluokan muodossa (Hudak ja Fasel 1992).

Listaus 9. Monad-tyyppiluokka (Hudak ja Fasel 1992).

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    (>>)   :: m a → m b → m b
    return :: a → m a
    fail   :: String → m a
    m >> k = m >>= \_ → k
```

Monadien olennaisimmat funktiot ovat return- ja bind-funktio (>>=). Funktioilla (>>=) ja (>>) ketjutetaan monadisia operaatioita. return-funktiolla laitetaan arvo monadin sisälle. (Hudak ja Fasel 1992.)

(>>=)-funktion toimintaa on helpompi ymmärtää esimerkin avulla. Funktio (>>=) on tyypiltään m a →(a → m b)→ m b. Funktio ottaa parametreinaan monadisen arvon a ja funktion. Lopputuloksena se palauttaa monadisen arvon b. Listaus 10 havainnollistaa (>>=)-funktion toimintaa ja do-notaatiota.

Listauksessa 10 on kolme kuvitteellista funktiota kysy, hae ja tulosta, jotka tekevät sivuvaikutuksellisia toimintoja IO-monadissa. ketjutettu määrittelee monadisen toiminnon, joka suorittaa peräkkäin kaikki kolme edellä mainituista funktioista siten, että edeltävän paluuarvo menee aina seuraavalle funktiolle parametriksi. ketjutetuDo on toiminnaltaan identtinen, mutta siinä on käytetty do-notaatiota.

Haskell-kielestä löytyy IO-monadin lisäksi monia muitakin monadeja. Esimerkiksi Identity-monadi on sivuvaikutukseton monadi. Se sopii käytettäväksi tilanteissa, joissa edellytetään monadia, mutta ei haluta erityisiä sivuvaikutuksia.

Listaus 10. Esimerkki bind-funktion toiminnasta ja do-notaatiosta.

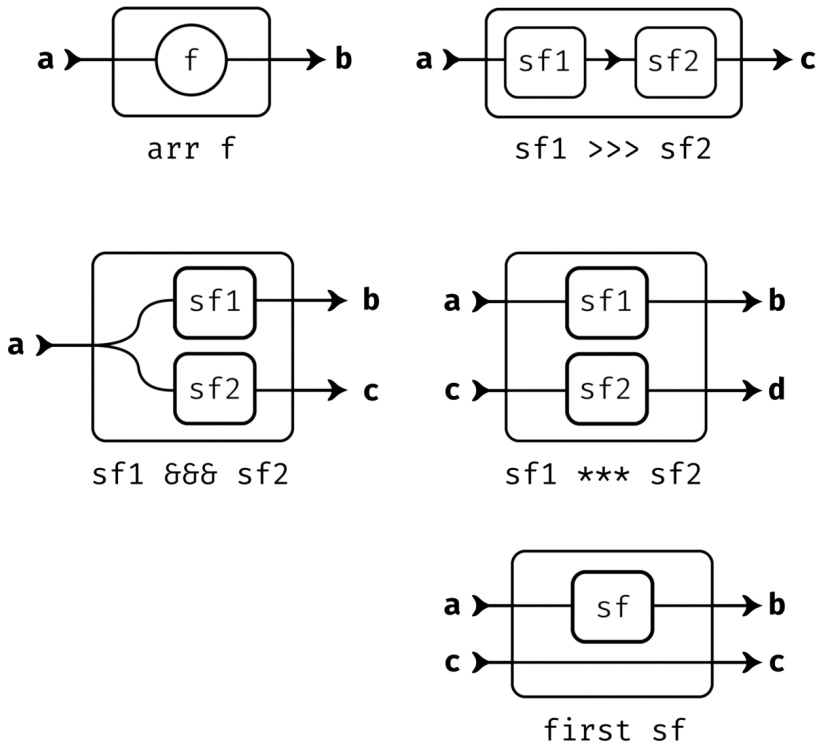
```
kysy    :: IO String
hae     :: String → IO String
tulosta :: String → IO ()

-- IO monadin tapauksessa:
(>>=)  :: IO a → (a → IO b) → IO b

ketjutettu :: IO ()
ketjutettu = kysy >>= hae >>= tulosta

ketjutettuDo :: IO ()
ketjutettuDo = do
    nimi ← kysy
    tieto ← hae nimi
    tulosta tieto
```

Nuolilla on yhtäläisyyksiä monadien kanssa. Nuolilla pystyy kuvaamaan toimintoja ja yleistä laskentaa abstraktilla tasolla, mutta monipuolisemmin kuin monadeilla (Hughes 2004). Nuolet on toteutettu Haskell-ohjelmointikieleen listauksessa 11 esitetyn tyyppiluokan muodossa. Sivuvaikutuksetonta laskentaa voi sisällyttää nuoliin `arr`-funktiolla, joka toimii monadien `return`-funktion tavoin. Nuolien ketjutus tehdään `(>>>)`-funktiolla, joka toimii monadien `(>>=)`-funktion tavalla (Hughes 2004). Funktiot `(***)`, `(&&&)` ja `first` ohjaavat laskentaa arvoparien avulla. Nuoliluokan funktioiden toimintaa on havainnollistettu kuviossa 2.

Listaus 11. `Arrow`-tyyppiluokka (Hughes 2004).

```
class Arrow arr where
    arr   :: (a → b) → arr a b
    (>>>)  :: arr a b → arr b c → arr a c
    (***) :: arr a b → arr c d → arr (a, c) (b, d)
    (&&&) :: arr a b → arr a c → arr a (b, c)
    first :: arr a b → arr (a, c) (b, c)
```

Paterson (2001) on kehittänyt uutta syntaksia nuolten käytön helpottamiseksi. Uutta syntaksia voi käyttää ainoastaan proc-notaation sisällä. Yksinkertaisin uudesta



Kuvio 2. `Arrow`-tyyppiluokan funktiot visualisoituna.

syntaksista on nuolen käyttö `a -< b`, jossa `a` on nuolta kuvaava lauseke ja `b` on lauseke, joka annetaan nuoleen syötteenä. Proc-notaatio on syntaktista sokeria ja se muunnetaan ohjelman käännön aikana normaaliksi Haskell-syntaksiksi (Paterson 2001).

Listauksen 12 esimerkissä voi nähdä monadien ja nuolien yhtäläisyyksiä. Esimerkin `addM`-funktio lisää yhteen kaksi monadista arvoa. `addA`-funktio kuvaa nuolta, joka antaa syötteensä eteenpäin kahdelle parametrina tulleelle nuolelle ja palauttaa niiden summan. `returnA`-funktiolla on sama rooli kuin monadien `return`-funktiolla.

Listaus 12. Esimerkki monadien ja nuolien yhtäläisyyksistä.

```
addM :: Monad m => m Int -> m Int -> m Int
addM a b = do
    x <- a
    y <- b
    return (x + y)
```

```
addA :: Arrow arr ⇒ arr b Int → arr b Int → arr b Int
addA f g = proc z → do
    x ← f ⤙ z

    y ← g ⤙ z

    returnA ⤙ x + y


returnA :: Arrow arr ⇒ arr a b
returnA = arr id
```

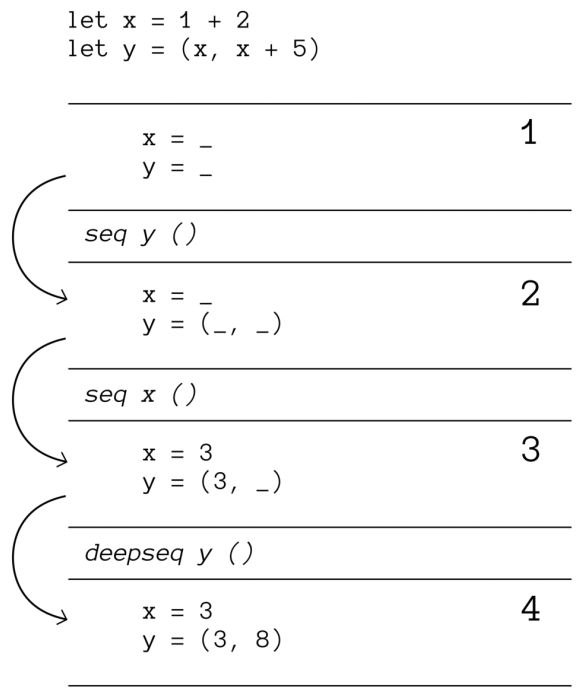
2.2.4 Laiska laskenta

Ohjelmointikielissä laiskuus tarkoittaa sitä, että lausekkeiden arvot lasketaan vasta kun niitä tarvitaan. Laiskan laskennan toimintaa havainnollistetaan kuviossa 3. Vaiheessa 1 `x` ja `y` ovat täysin laskemattomia lausekkeita, jota on kuvattu alaviivalla. Vaiheiden väleissä kutsutaan `seq`-funktiota antamalla sille kulloinkin tilanteeseen sopivat parametrit.

Esimerkissä vaiheiden 1 ja 2 välissä parametreina annetaan `y` ja tyhjä arvo. `seq`-funktio laskee ensimmäisen parametrinsa *weak head normal form* (WHNF) -muotoon ja palauttaa sen jälkeen jälkimmäisen parametrinsa (Marlow 2013). WHNF-muotoon laskeminen tarkoittaa sitä, että lausekkeesta lasketaan vain minimaalinen määrä. Tietotyyppien tapauksessa lasketaan vain konstruktorin verran, jolloin tiedetään `y`:n rakenne, mutta ei vielä sisältöä.

Vaiheen 2 ja 3 välissä `seq`-funktiota kutsutaan parametreilla `x` ja tyhjä arvo. Lausekkeen `x` arvoksi tulee 3. Lausekkeen `y` arvosta on nyt laskettu arvoparin ensimmäinen arvo, mutta toisena arvona on vielä laskematon lauseke. Lausekkeen `y` arvoja saatetaan tarvita eriaikaisesti. Tämän vuoksi lausekkeen `y` arvo lasketaan loppuun vasta kun arvoparin kumpaakin arvoa on tarvittu.

Lausekkeen laskennan voi tarvittaessa pakottaa loppuun asti `deepseq`-funktiolla, jonka vaikutus näkyy vaiheessa 4. Loppuun asti laskettu lauseke on *normal form*



Kuvio 3. Esimerkki laiskan laskennan etenemisestä.

-muodossa (Marlow 2013). `deepseq`-funktion käyttö edellyttää, että laskettavalle arvolle on olemassa `NFData`-tyyppiluokan instanssi. `NFData`-tyyppiluokka on esitetty listauksessa 13. Tyyppiluokan `rnf`-funktio on nimeltään *reduce to normal form*. `rnf`-funktiolle löytyy valmiina oletustoteutus, joka käyttää `seq`-funktiota.

Listaus 13. `NFData`-tyyppiluokan määritelmä (Marlow 2013).

```
class NFData a where
    rnf :: a → ()
    rnf a = a `seq` ()
```

2.3 Reaktiivinen funktio-ohjelmointi

Reaktiivinen funktio-ohjelmointi on lähtöisin Fran-ohjelmointikielestä, jonka on luonut Elliott ja Hudak (1997). Heidän tarkoituksenaan oli toteuttaa reaktiivisia systeemejä käyttämällä apuna funktio-ohjelmointia.

Reaktiivisen funktio-ohjelmoinnin toteutustavat voidaan jakaa kahteen kategoriaan. Ensimmäinen niistä on klassinen reaktiivinen funktio-ohjelmointi, joka keskittyy ajasta riippuviin funktioihin, eli signaaleihin, ja toinen on nuolipohjainen reaktiivinen funktio-ohjelmointi, joka keskittyy signaalifunktioihin, jotka muuntavat signaaleja toisenlaisiksi (Perez, Bärenz ja Nilsson 2016).

2.3.1 Signaalit ja signaalifunktiot

Sculthorpe ja Nilsson (2010) ovat tehneet määritelmiä, joiden mukaan reaktiivisen funktio-ohjelmoinnin konsepteja voi mallintaa seuraavalla tavalla: signaaleja voi mallintaa funktioina, jotka ottavat parametrina ajan ja palauttavat arvon, ja aikaa kuvataan jatkuvana positiivisena reaalilukuna. Edellä mainitut määritelmät ovat esitettynä määritelmässä (2.1), jossa tyyppiparametri α kuvastaa signaalin palauttaman arvon tyyppiä.

$$\begin{aligned} Time &\approx \{\, t \in \mathbb{R} \mid t \geqslant 0 \,\} \\ Signal\ \alpha &\approx Time \to \alpha \end{aligned} \tag{2.1}$$

Signaali voi kuvastaa mitä tahansa ajasta riippuvaa asiaa. Esimerkiksi pelissä voi olla signaali, joka antaa hiiren kursorin sijainnin tietyllä ajanhetkellä. Signaali voisi kuvata myös jonkin pelissä olevan objektin tilaa, joka muuttuu ajan myötä.

Edellä kuvatun signaalin idean käytännössä toteuttamiseen vaaditaan rajoitteita. Signaalin historiaa tulee voida selata vain rajallisen määrän verran tilavuotojen rajoittamiseksi ja sen lisäksi signaalien tulee toimia ajan suhteen kausaalisesti. Kausaalisuudella tarkoitetaan, että ne eivät voi riippua toisten signaalien arvoista, eikä omista arvoistaan, joiden ajankohta on tulevaisuudessa (Perez, Bärenz ja Nilsson 2016; Sculthorpe ja Nilsson 2010).

Signaalifunktioita voi mallintaa funktioina, jotka ottavat parametrina signaalin ja palauttavat signaalin. Tämä on esitetty määritelmässä (2.2).

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta \tag{2.2}$$

Yksinkertainen toteutus signaalifunktiolle on esitetty listauksessa 14. Tyyppiparametri `a` kuvastaa sisääntulevan signaalin sisältöä ja vastaa määritelmässä (2.2) olevaa tyyppiparametria α ja tyyppiparametri `b` kuvastaa ulostulevan signaalin sisältöä ja vastaa tyyppiparametria β. `SF`-tyyppi sisältää funktion, joka ottaa parametrina tyyppiä `a` olevan arvon ja palauttaa tyyppiä `b` olevan arvon, sekä uuden signaalifunktion, joka määrittää, millä tavalla signaalifunktio toimii, kun sen arvoa lasketaan seuraavan kerran. Tämä toteutus on hyvin yksinkertainen, eikä ota kantaa ajanhallintaan. Ajan voi esimerkiksi määritellä etenemään vakiomäärän verran jokaisen signaalin arvon laskemisen välissä.

Listaus 14. Signaalifunktion naiivi toteutus.

```
newtype SF a b = SF (a → (b, SF a b))
```

Yampa-kirjasto on Haskell-ohjelmointikielen sisälle rakennettu täsmäkieli, jolla kuvataan reaktiivisia systeemejä (Courtney, Nilsson ja Peterson 2003). Yampa-kirjasto on toteutettu ideatasolla listauksessa 14 kuvatulla tavalla, mutta siinä ajanhallinta on otettu huomioon toteutuksessa ja toteutus on huomattavasti monimutkaisempi runsaan optimoinnin takia. Listauksen 14 ja Yampa-kirjaston signaalifunktiot toimivat täysin sivuvaikutuksettomasti ja ovat viitteellisesti läpinäkyviä, jolloin signaalifunktiot eivät voi vaikuttaa toisiinsa millään tavalla ilman eksplisiittisiä kytkentöjä. Tämän seurauksena kahden toisistaan riippumattoman signaalifunktion arvon laskemisen järjestyksellä ei ole mitään merkitystä.

Yampa-kirjasto ja vastaavat toteutukset eivät ole ongelmattomia. Näissä ajanhallinta on toteutettu globaalilla tasolla, jolloin ei ole mahdollista, että mahdolliset alijärjestelmät toimisivat nopeammin tai hitaammin Perez, Bärenz ja Nilsson (2016). Pelejä toteuttaessa saatetaan esimerkiksi haluta, että fysiikkamoottorin päivitys tapahtuisi eri nopeudella kuin muun pelilogiikan. Lisäksi signaalifunktioiden sivuvaikutuksettomuuden takia käyttäjän kanssa kommunikointi tapahtuu korkeimmalla mahdollisella tasolla. Tällöin joudutaan hakemaan kaikkien syöttölaitteiden tila ja

signaalifunktiosta saadaan ulos yksi suuri tietorakenne, joka kuvastaa koko ohjelman sen hetkistä tilaa. Sivuvaikutuksettomuus tekee myös debuggaamisesta haastavaa, koska ainoa keino saada tietoa ulos yksittäisestä signaalifunktiosta on määritellä sen ulostulon tyyppi sellaiseksi, että se sisältää myös debuggaamiseen vaaditut tiedot (Perez, Bärenz ja Nilsson 2016).

Näiden ongelmien korjaamiseen Perez, Bärenz ja Nilsson (2016) ovat esitelleet listauksessa 15 esitetyn monadisen virtamuuntimen, joka voi ulostuloaan laskiessaan suorittaa monadisia toimintoja. Tyyppimuuttuja `m` kuvastaa mikä monadi on kyseessä ja tyyppimuuttujat `a` ja `b` kuvastavat sisääntulon ja ulostulon tyyppiä.

Listaus 15. Monadinen virtamuunnin (Perez, Bärenz ja Nilsson 2016).

```
newtype MSF m a b
```

Myöhemmässä vaiheessa tutkielmaa keskitytään vain Netwire-kirjastoon, jonka `Wire`-tyyppi käyttäytyy monadisen virtamuuntimen tavoin. Uusia signaalifunktioita voi luoda signaalifunktioiden luontifunktiolla, joista kerrotaan luvussa 2.3.2. Netwire-kirjastossa signaalifunktioita kuvastaa `Wire`-tyyppi, jonka määritelmä ilman konstruktoreita on esitetty listauksessa 16.

Listaus 16. `Wire`-tyypin määritelmä (Söylemez 2017).

```
data Wire s e m a b
```

`Wire`-tyypillä on viisi tyyppiparametria. Ensimmäinen tyyppiparametri `s` kuvastaa paljonko aikaa on kulunut viimeisimmästä evaluaatiosta. Netwire-kirjaston signaalifunktiot voivat halutessaan lopettaa arvojen tuottamisen kokonaan. Toinen tyyppiparametri `e` kuvastaa minkä tyyppinen arvo annetaan siinä tapauksessa, kun signaalifunktion varsinainen toiminta lakkaa ja arvojen tuottaminen päättyy. Monet signaalifunktioiden luontifunktiot vaativat, että tyypin `e` täytyy olla monoidi. Netwire-kirjaston signaalifunktiot voivat tehdä myös monadisia toimintoja. Kolmas tyyppiparametri `m` kuvaa mikä monadi on kyseessä. Neljäs tyyppiparametri `a` kuvastaa signaalifunktion syötteen tyyppiä, joka vastaa määritelmän (2.2) tyyppimuuttujaa α. Viides tyyppiparametri `b` kuvastaa signaalifunktion ulostulon tyyppiä, joka vastaa määritelmän (2.2) tyyppimuuttujaa β. Netwire-kirjasto määrittelee myös tyyppisy-

nonyymin nimeltään `WireP`, joka on esitetty listauksessa 17. `WireP`-tyyppi määrittää sivuvaikutuksista vapaan signaalifunktion rajoittamalla monadin identiteettimonadiksi.

Listaus 17. `WireP`-tyypin määritelmä (Söylemez 2017).

```
type WireP s e = Wire s e Identity
```

2.3.2 Signaalifunktioiden luominen

Netwire-kirjasto tarjoaa useita funktioita signaalifunktioiden luomiseen. Tyypillisesti nämä funktiot ottavat parametrina toisen funktion, joka määrittelee signaalifunktion toiminnan. Esimerkiksi listauksessa 18 oleva `mkSF`-funktio ottaa parametrina tavallisen funktion ja muuttaa sen signaalifunktioksi.

Listauksessa on myös signaalifunktion luontifunktio `mkSF_`. Tällä luodaan signaalifunktio, jolla on sisäinen tila. `mkSF_`-funktion ensimmäinen parametri on signaalifunktion toimintaa kuvaava funktio. Sisäinen tila näkyy siinä, että parametrina annettu funktio palauttaa mielivaltaisen signaalifunktion, joka määrittelee miten signaalifunktio käyttäytyy seuraavaa arvoa laskiessa. `mkGen`-funktio toimii `mkSF_`-funktion tavoin, mutta se pystyy lakkauttamaan -arvojen tuottamisen.

Listaus 18. Esimerkkejä Netwire-kirjaston signaalifunktioiden luontifunktioista (Söylemez 2017).

```
mkSF_ :: (a → b) → Wire s e m a b

mkSF  :: Monoid s ⇒ (s → a → (b, Wire s e m a b)) → Wire s e m a b

mkGen :: (Monad m, Monoid s)
      ⇒ (s → a → m (Either e b, Wire s e m a b)) → Wire s e m a b
```

Esimerkiksi ajan suhteen integroivan signaalifunktion tekoon tarvittaisiin signaalifunktion luontifunktiota, joka säilyttää sisäisen tilan. Listauksessa 19 on esitetty signaalifunktio `integraali`, joka integroi syötteensä ajan suhteen. Syöte ja tuotettu arvo on rationaaliluku `Fractional`-tyyppiluokkarajoitteen takia. Apumuuttujaan

`dt` lasketaan kulunut aika, ja signaalifunktio palauttaa integroimisvakion `c` ja jatkaa toimintaansa kutsumalla itseään siten, että seuraavaan integroimisvakioon on lisätty signaalifunktion syöte `x` kerrottuna kuluneella ajalla `dt`.

Listaus 19. `integraali`-signaalifunktio.

```
integraali :: (Fractional a, HasTime a s) ⇒ a → Wire s e m a a
integraali c = mkSF $ \s x → let dt = realToFrac (dtime s)
                             in (c, integraali (c + x * dt))
```

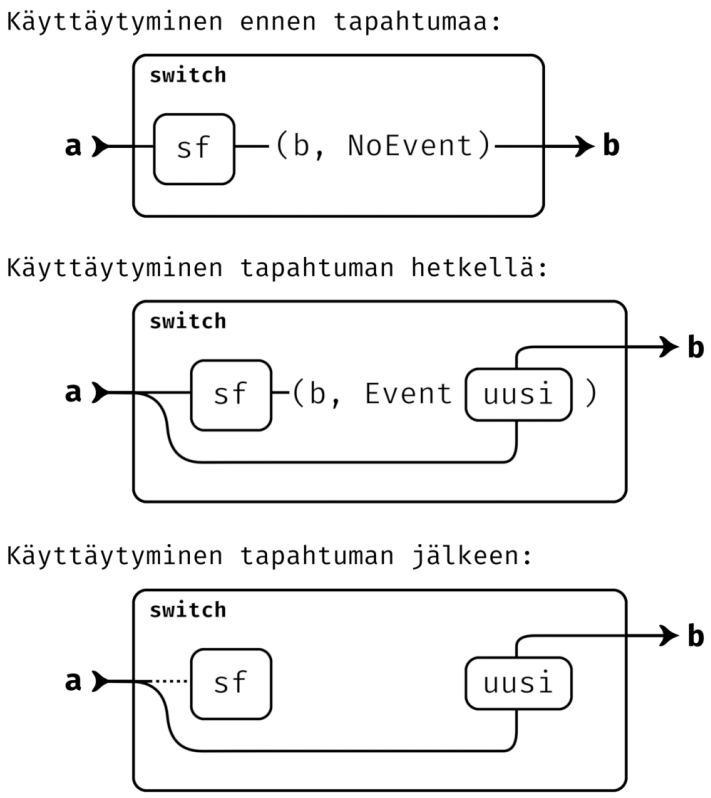
2.3.3 Signaalifunktion toiminnan muuttaminen

Suurin osa reaktiivisen funktio-ohjelmoinnin toteutuksista tukee rakenteellista dynaamisuutta. Rakenteellisella dynaamisuudella tarkoitetaan sitä, että ohjelman suorituksen aikana on mahdollista luoda uusia signaalifunktioita ja signaalifunktioiden keskinäisistä riippuvuuksista muodostuva verkko voi muuttua. Muutoksia signaalifunktioverkon rakenteeseen kutsutaan rakenteellisiksi kytkimiksi (engl. *structural switch*). (Sculthorpe ja Nilsson 2010.)

Netwire-kirjastosta monenlaisia kytkimiä, joilla voi muuttaa signaalifunktion toimintaa (Söylemez 2017). Yksinkertaisin kytkimistä on kuviossa 4 havainnollistettu `switch`-funktio, jonka tyyppimääritelmä on esitetty listauksessa 20. `switch`-funktio toimii siten, että se ottaa parametrina signaalifunktion `sf` tuottaen arvoparin, joka koostuu `b`-tyyppisestä arvosta ja mahdollisesta tapahtumasta, joka sisältää signaalifunktion `uusi`. Tapahtuman tullessa signaalifunktio alkaa heti toimimaan tapahtuman sisältämän `uusi`-signaalifunktion mukaisesti.

Listaus 20. `switch`-funktion tyyppimääritelmä (Söylemez 2017).

```
switch :: (Monad m, Monoid s)
       ⇒ Wire s e m a (b, Event (Wire s e m a b))
       → Wire s e m a b
```



Kuvio 4. `switch`-funktion toiminta.

2.3.4 Peliobjektien kuvaaminen signaalifunktioilla

Courtney, Nilsson ja Peterson (2003) ovat tutkineet peliobjektien mallintamista signaalifunktioilla. Heidän tutkimuksensa perustuu Yampa-kirjastoon, mutta samat asiat ovat sovellettavissa Netwire-kirjastoon.

Listauksen 21 esimerkissä on esitetty miten Netwire-kirjastolla voi mallintaa peliobjekteja signaalifunktioilla. Courtney, Nilsson ja Peterson (2003) tekemän tutkimuksen mukaisesti listauksessa 21 peliobjekti on signaalifunktio, joka ottaa syötteenä `ObjectInput`-tyyppisen arvon ja tuottaa `ObjectOutput`-tyyppisen arvon. Objektille tulevaan syötteeseen sisältyy käyttäjän syöte, esimerkiksi näppäimistön ja hiiren tila, sekä tiedot pelin tilasta, jotka vaikuttavat objektin käyttäytymiseen. Objektia kuvaavan signaalifunktion tuottama arvo koostuu objektin tilasta ja tapahtuman muodossa uusista objekteista, jotka objekti on luonut.

Esimerkissä on esitetty kuvitteellinen peli, jossa on vihollisen avaruusalus, joka lentää vakionopeudella ja ampuu neljän sekunnin välein ammuksia. Aluksen sijainti on määritetty olemaan nopeuden integraali ajan suhteen ja integrointivakiona on aloitussijainti. Integrointi toimii listauksessa 19 olevan `integraali`-funktion mukaisesti, mutta lukujen sijaan integroidaan vektoreita. Objektille tulevaan syötteeseen sisältyy tieto pelaajan ammuksien sijainneista. Alus muuttuu räjähdystä kuvaavaksi signaalifunktioksi törmätessään pelaajan ampumiin ammuksiin.

Listaus 21. Esimerkki peliobjektien mallintamisesta signaalifunktioilla

```
-- Objektin tilaan sisältyy vektori, joka kuvastaa sen sijaintia.
data ObjectState = ObjectState Vector

-- Objekti on sivuvaikutukseton signaalifunktio.
type Object = WireP GameSession () ObjectInput ObjectOutput

-- Signaalifunktio, jolla integroidaan vektoria ajan suhteen.
integral :: Vector → Wire s e m Vector Vector
integral = ...

-- Räjähdystä kuvaava peliobjekti. Parametrina annetaan aloitussijainti.
explosion :: Vector → Object
explosion = ...

enemyBullet = Vector → Object
enemyBullet = ...

-- Vihollisen alus. Parametreina aloitussijainti ja -nopeus.
enemyShip :: Vector → Vector → Object
enemyShip p v = switch $ proc input → do
    -- Aluksen sijainti on nopeuden integraali.
    shipPosition ← integral p -< v
    let objState = ObjectState shipPosition

    -- Neljän sekunnin välein tulee tapahtuma, joka sisältää
    -- aluksen ampumaa ammusta kuvaavan signaalifunktion.
    newBulletEvent ← periodic 4.0 <<< arr enemyBullet -< shipPosition
```

```
    -- Alus muuttuu räjähdykseksi törmätessään pelaajan ammuksiin.
    let switchEvent = if objState `intersects` playerBullets input
                          then Event (explosion shipPosition)
                          else NoEvent
    returnA -< (ObjectOutput objState newBulletEvent, switchEvent)
```

2.3.5 Signaalifunktioiden evaluaatio

Netwire-kirjaston signaalifunktioiden arvoja pystyy laskemaan kirjastossa olevan `stepWire`-funktion avulla, jonka tyyppimääritelmä löytyy listauksesta 22. `stepWire`-funktio ottaa parametreina päivitettävän signaalifunktion, `s`-tyyppisen kuluvaa aikaa kuvastavan arvon ja `Either e a` -tyyppisen arvon, joka toimii signaalifunktion syötteenä. Syötteenä `Right`-tyyppikonstruktorilla annetaan `a`-tyyppinen syöte, jonka signaalifunktio työstää `b`-tyyppiseksi ulostuloksi. `Left`-tyyppikonstruktorilla annetaan `e`-tyyppinen arvo, joka aiheuttaa sen, että signaalifunktio lakkaa tuottamasta arvoja.

Listaus 22. Funktion `stepWire` tyyppi (Söylemez 2017).

```
stepWire :: Monad m
         => Wire s e m a b -> s -> Either e a
         -> m (Either e b, Wire s e m a b)
```

Signaalifunktio voi arvoaan laskiessaan suorittaa monadisia toimintoja, jonka takia `stepWire`-funktio palauttaa arvoparin `m`-monadiin käärittynä. Arvopari koostuu signaalifunktion ulostulosta ja uudesta signaalifunktiosta, joka kuvaa signaalifunktion uutta tilaa. Signaalifunktion tulostulo on onnistuneesti tuotettuna `b`-tyyppinen arvo tai `e`-tyyppinen arvo, joka tarkoittaa sitä, että signaalifunktio on lakannut tuottamasta arvoja.

Peleissä, joissa päivitettäviä peliobjekteja on useita, peliobjekteja kuvaavia signaalifunktioita on tarvetta säilyttää jonkinlaisessa kokoelmassa. Jotta kaikkia kokoelman signaalifunktioita voi päivittää, täytyy jokaiselle signaalifunktiolle kutsua `stepWire`-funktiota ja ottaa talteen signaalifunktion tuottama arvo sekä signaalifunktion

uusi tila. Tästä ideasta on esitetty mallitoteutus listauksessa 23. `stepWire`-funktiossa käytetty apufunktio `step` on lyhennetty versio `stepWire`-funktiosta, jolle on annettu valmiina aika (`dt`) ja syöte (`Right inp`). Funktio `mapM` kutsuu funktiota `step` jokaiselle `wires` listassa olevalle signaalifunktiolle. Signaalifunktiot, jotka lakkaavat tuottamasta arvoja pudotetaan pois, jolloin `filteredOutput` listaan jäävät aktiiviset signaalifunktiot ja niiden tuottamat arvot.

Listaus 23. Signaalifunktiokokoelman evaluaatio

```
stepWires :: Monad m
          ⇒ s → a → [Wire s e m a b] → m [(b, Wire s e m a b)]
stepWires dt inp wires = do
    let step w = stepWire w dt (Right inp)
    output ← mapM step wires
    let filteredOutput = [(o, w') | (Right o, w') ← output]
    return filteredOutput
```

Listauksessa 23 esitetyn funktion ongelmana on se, että sitä on hankala käyttää osana muita signaalifunktioita. Signaalifunktioita ohjelmoidessa ajanhallinta tapahtuu implisiittisesti, eikä aikaa, eli `s`-tyypin parametria, ole käytännössä mahdollista kuljettaa eksplisiittisenä parametrina `stepWires`-funktiolle. Ongelman voi ratkaista muotoilemalla `stepWires`-funktio signaalifunktioksi, joka ottaa syötteenä listan päivitettävistä signaalifunktioista ja antaa ulostulona aktiivisten signaalifunktioiden tuottamat arvot sekä uudet tilat. `stepWires`-funktio on esitetty signaalifunktion muodossa listauksessa 24. Listauksessa oleva `mkGen`-signaalifunktion luontifunktio edellyttää ylimääräisen `Monoid s` -tyyppiluokkarajoituksen.

Listaus 24. Signaalifunktiokokoelman evaluaatio muotoiltuna signaalifunktioksi

```
stepWires :: (Monad m, Monoid s)
          ⇒ Wire s e m ([Wire s e m a b], a) [(b, Wire s e m a b)]
stepWires =
    mkGen $ \dt (wires, inp) → do
        let step w = stepWire w dt (Right inp)
        output ← mapM step wires
        let filteredOutput = [(o, w') | (Right o, w') ← output]
        return (Right filteredOutput, stepWires)
```

2.4 Rinnakkaislaskenta

Tässä luvussa esitetään, kuinka rinnakkaislaskentaa voidaan toteuttaa Haskell-ohjelmointikielellä ja kuinka lista voidaan päivittää rinnakkaisesti. Listan rinnakkaisella päivityksellä tarkoitetaan tietyn kuvausfunktion kutsumista listan jokaiselle alkiolle siten, että jokainen funktiokutsu suoritetaan rinnakkaisesti. Lopputuloksena syntyy uusi lista, jossa on aikaisemman listan alkiot syötettynä kuvausfunktion läpi. Luvun lopussa esitellään Amdahlin laki, joka antaa teoreettisen maksimaalisen ylärajan suorituskyvyn parantumiselle, joka voidaan saavuttaa rinnakkaislaskennalla.

Rinnakkaisuuteen liittyy yhtäaikaisuus (engl. *concurrency*) ja rinnakkaislaskenta (engl. *parallelism*). Yhtäaikaisuus on tekniikka, jolla ohjelma rakennetaan siten, että se voi olla yhtäaikaisesti interaktiossa esimerkiksi käyttäjän, tietokantojen tai muiden ulkopuolisten asioiden kanssa. Tämä toteutetaan usein säikeillä, joiden suoritus tapahtuu lomittain tai samaan aikaan. Rinnakkaislaskenta eroaa yhtäaikaisuudesta siten, että siinä keskitytään hyödyntämään prosessorin useita ytimiä tehokkaan laskennan saavuttamiseksi. Rinnakkaislaskennan ainoana tavoitteena on saada laskenta suoritettua nopeammin. (Marlow 2013.)

2.4.1 Rinnakkaislaskenta Haskell-ohjelmointikielellä

GHC-kääntäjällä käännettyihin Haskell-ohjelmiin linkitetään mukaan GHC-ajoympäristö (engl. *GHC runtime system*, RTS), jonka päällä Haskell-ohjelmaa ajetaan (GHC Team 2015). RTS tukee miljoonia kevyitä säikeitä, joita käyttöjärjestelmän säikeet ottavat suoritettavakseen (Marlow, Peyton Jones ja Singh 2009). Käyttöjärjestelmän säikeiden ja RTS:n kevyiden säikeiden erottamiseksi RTS:n kevyitä säikeitä nimitetään tästä eteenpäin Haskell-säikeiksi. Jokaiselle käyttöjärjestelmän säikeelle on olemassa laskentakonteksti (engl. *Haskell Execution Context*, HEC), joka sisältää kaikki tarvittavat tiedot, joita käyttöjärjestelmän säie tarvitsee Haskell-säikeiden suorittamiseen (Marlow, Peyton Jones ja Singh 2009).

Kipinä on laskematon lauseke, joka odottaa, että jokin vapaista säikeistä ottaa sen laskettavaksi (Marlow, Peyton Jones ja Singh 2009). Marlow, Peyton Jones ja Singh

(2009) ovat kirjoittaneet, että Trinderin, Hammondin, Loidlin, ja Peyton Jonesin (1998) mukaan rinnakkaislaskentaa tarvitsevat funktiot voidaan toteuttaa yhdistelemällä listauksen 25 kahta funktiota.

Listaus 25. Rinnakkaislaskentakombinaattorit (Marlow, Peyton Jones ja Singh 2009).

```
par  :: a → b → b
pseq :: a → b → b
```

Listauksessa 25 esitetty `par`-funktio ottaa parametrina `a`-tyyppisen arvon, josta luodaan kipinä (engl. *spark*) (Marlow, Peyton Jones ja Singh 2009). Tämän jälkeen funktio alkaa laskemaan toista parametriaan (Marlow, Peyton Jones ja Singh 2009). Esimerkiksi lausekkeen lausekkeen `par a b` arvo on suoraan `b`, mutta ensimmäisestä parametrista `a` on ennen toisen parametrin laskentaa luotu kipinä, jolloin `a` ja `b` voidaan laskea rinnakkaisesti.

`pseq`-funktio on toimii määritelmän (2.3) mukaisesti. Symboli ⊥ kuvastaa lauseketta, jonka laskeminen ei pääty onnistuneesti, johtuen virheestä tai loputtomasta rekursiosta. `par`-funktion lisäksi tarvitaan `pseq`-funktiota ohjaamaan sarjassa suoritettavia laskuja. `pseq`-funktio laskee ensimmäisen parametrinsa *weak head normal form* -muotoon ja aloittaa sen jälkeen laskemaan jälkimmäistä parametriaan. (Marlow, Peyton Jones ja Singh 2009.)

$$\begin{aligned} \texttt{pseq a b} &= \bot, \quad \textit{jos } \texttt{a} = \bot \\ &= \texttt{b}, \quad \textit{muussa tapauksessa} \end{aligned} \tag{2.3}$$

Marlow, Peyton Jones ja Singh (2009) esittävät, että listan rinnakkainen päivitys voidaan toteuttaa listauksessa 26 esitetyllä tavalla. `parMap`-funktio toimii siten, että lausekkeesta `f x` luodaan kipinä, jonka jälkeen lasketaan lausekkeen `parMap f xs` arvo ennen uuden listan `y:ys` palauttamista. `pseq`-funktiolla varmistetaan, että laskenta tapahtuu oikeassa järjestyksessä.

Listaus 26. Toteutus parMap-funktiolle (Marlow, Peyton Jones ja Singh 2009).

```
parMap f []     = []
parMap f (x:xs) = y `par` (ys `pseq` y:ys)
    where y  = f x
          ys = parMap f xs
```

2.4.2 Eval-monadi

Eval-monadilla voi kuvata yleisellä tasolla laskentaa. Listauksessa 27 esitetyt funktiot rpar ja rseq kuvaavat rinnakkain ja sarjassa suoritettavia laskennan osia (Marlow 2013). Funktiot rpar ja rseq ovat Eval-monadin apufunktioita, jotka vastaavat toiminnaltaan listauksessa 25 esiteltyjä funktioita par ja pseq. Eval-monadi on määritetty Haskell-moduulissa Control.Parallel.Strategies ja sen avulla voi kehittää evaluaatiostrategioita, joista kerrotaan lisää luvussa 2.4.3.

Listaus 27. Eval-monadin määritelmät (Marlow 2013).

```
data Eval a
instance Monad Eval

runEval :: Eval a → a

rpar :: a → Eval a
rseq :: a → Eval a
```

Eval-monadia voi käyttää monadien tavoin do-notaatiolla. Esimerkiksi luvussa 2.4.1 käytetyillä määritelmillä lauseketta x `par` (y `pseq` (x + y)) vastaava toteutus voidaan tehdä Eval-monadilla listauksessa 28 esitetyllä tavalla.

Listaus 28. Esimerkki Eval-monadin käytöstä.

```
runEval $ do
    a ← rpar x
    b ← rseq y
    return (a + b)
```

2.4.3 Listan rinnakkaistettu päivitys evaluaatiostrategioilla

Evaluaatiostrategioilla voidaan tehdä rinnakkaislaskentaa hyödyntävästä koodista modulaarisempaa erottamalla algoritmi rinnakkaisuudesta. Tämä mahdollistaa sen, että koodin voi rinnakkaistaa eri tavoilla pelkästään vaihtamalla evaluaatiostrategiaa. (Marlow 2013.)

`Strategy`-tyyppi kuvastaa evaluaatiostrategiaa ja se on määritetty Haskell-moduulissa `Control.Parallel.Strategies` listauksessa 29 esitetyllä tavalla.

Listaus 29. Strategy-tyypin määritelmä (Hackage 2017a).

```
type Strategy a = a → Eval a
```

Esimerkkejä evaluaatiostrategioista ovat listauksessa 27 esitetyt funktiot `rseq` ja `rpar`. Näiden lisäksi on olemassa evaluaatiostrategiat `r0`, joka ei suorita laskentaa ollenkaan ja `rdeepseq`, joka laskee parametrinsa täysin, eli *normal form* -muotoon.

Mikäli evaluaatiostrategioita haluaan hyödyntää listan rinnakkaistetussa päivityksessä, tarvitaan tämän toteuttamiseen apufunktioita. Listauksessa 30 esitetty funktio `parList`, joka laskee rinnakkain kaikki listan alkiot `evalList`-funktion avulla. Listauksessa oleva funktio `rparWith` on määritetty `Control.Parallel.Strategies`-moduulissa. Funktiolla `rparWith` yhdistetään `rpar`-strategia toisen, erityyppisen strategian kanssa. `using`-funktiolla selkeytetään koodia. Esimerkiksi lauseke `x ‘using‘ strat` tarkoittaa sitä, että `x`:n arvo lasketaan käyttäen `strat`-evaluaatiostrategiaa.

Listaus 30. Toteutus `parList` ja `evalList` -funktioille (Hackage 2017a).

```
parList :: Strategy a → Strategy [a]
parList strat = evalList (rparWith strat)

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

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

Listauksessa 30 esitettyjen apufunktioiden avulla voidaan luoda toteutus `parMap`-funktiolle. Toteutus on esitetty listauksessa 31. Funktio `parMap` kutsuu funktiota `f` jokaiselle listan alkiolle rinnakkaisesti. Jokainen listan alkio lasketaan `rseq`-evaluaatiostrategialla *weak head normal form* -muotoon. Vaihtamalla `rseq` tilalle esimerkiksi `rdeepseq`, voidaan kaikki alkiot laskea *normal form* -muotoon.

Listaus 31. Toteutus `parMap`-funktiolle evaluaatiostrategian kanssa (Marlow 2013).

```
parMap :: (a → b) → [a] → [b]
parMap f xs = map f xs `using` parList rseq
```

2.4.4 Lohkottaminen

Lohkottaminen (engl. *chunking*) on yleisesti käytetty tekniikka, jolla vältetään laskennan jakamista liian pieniksi paloiksi pilkkomalla lista samankokoisiin lohkoihin, joiden alkiot lasketaan kerralla (Marlow, Peyton Jones ja Singh 2009; Marlow 2013). Jokaista lohkoa kohden luodaan vain yksi kipinä, jolla evaluoidaan kokonainen lohko listasta yksittäisen listan elementin sijaan (Marlow, Peyton Jones ja Singh 2009). Lohkottaminen saattaa parantaa suorituskykyä silloin, kun listan päivityksessä käytetty kuvausfunktio on laskennallisesti hyvin kevyt, tai kun listan alkioiden määrä on niin suuri, ettei jokaista alkiota kohden kannata luoda kipinää (Marlow 2013).

Lohkottamisen toteuttamiseen `Control.Parallel.Strategies`-moduuli tarjoaa valmiina `parListChunk`-funktion, jonka tyyppimääritelmä on esitetty listauksessa 32. `parListChunk`-funktion tyyppimääritelmä eroaa listauksessa 30 esitetyn funktion `parList` tyyppimääritelmästä siten, että `parListChunk`-funktiolle annetaan ensimmäisenä parametrina yksittäisen lohkon koko kokonaislukuna.

Listaus 32. `parListChunk`-funktion tyyppimääritelmä (Hackage 2017a).

```
parListChunk :: Int → Strategy a → Strategy [a]
```

Lohkottamisen voi ottaa käyttöön esimerkiksi listauksessa 31 olevaan `parMap`-funktioon. Lohkottamisen toteuttamiseen riittää lisätä yksittäisen lohkon alkioiden määrä parametriksi ja vaihtaa `parList`-funktion tilalle `parListChunk`, kuten listauksessa 33 on esitetty.

Listaus 33. `parMap`-funktio lohkottamisen kanssa.

```
parMap :: Int → (a → b) → [a] → [b]
parMap chunkSize f xs = map f xs `using` parListChunk chunkSize rseq
```

2.4.5 Amdahlin laki

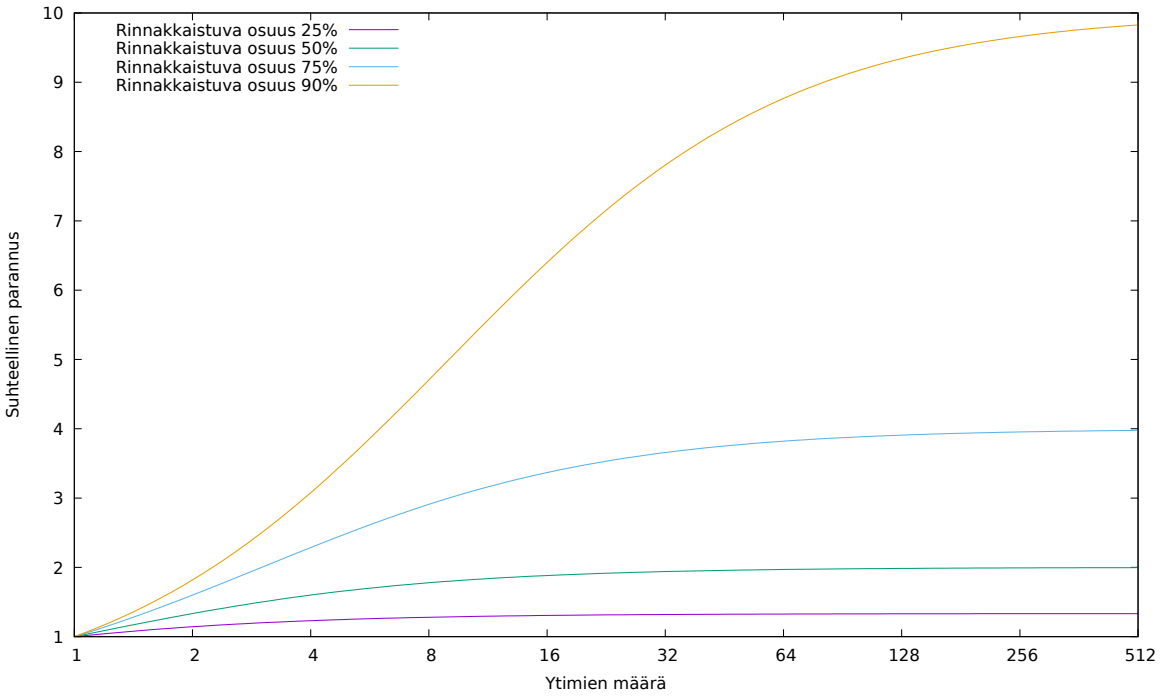
Aiempien tutkimusten (Amdahl 1967, 2013; Hill ja Marty 2017) mukaan suorituskykyä ei pysty parantamaan rajattomasti kasvattamalla prosessorin ytimien määrää. Alkuperäinen Ahmdalin laki (Amdahl 1967) ja sen myöhemmät parannukset (Gustafson 1988) ovat keinoja laskea suorituskyvyn parantumisen teoreettinen yläraja tilanteissa, joissa osa koodista voidaan suorittaa rinnakkaistettuna.

Amdahlin laki voidaan esittää muodossa (Hill ja Marty 2017)

$$parannus(f,s) = \frac{1}{(1-f) + \frac{f}{s}}, \tag{2.4}$$

jossa f on osuus koodista, joka voidaan rinnakkaistaa ja s on rinnakkaistetun osan suhteellinen parannus, jolloin *parannus* itsessään kuvaa parannusta koko ohjelman suorituskykyyn.

Kuviossa 5 on havainnollistettu yhtälössä (2.4) olevaa f parametria, eli rinnakkaistuvan osuuden määrää ja s parametria, eli rinnakkaistuvan osan suhteellista parantumista. Rinnakkaistuvan osuuden suorituskyvyn parantuminen oletetaan olevan suoraan verrannollinen ytimien määrään. Parametrin s lähestyessä ääretöntä suorituskyvyn parannus lähestyy raja-arvoa $1/(1-f)$ (Hill ja Marty 2017).



Kuvio 5. Amdahlin laki

2.5 Suorituskyvyn parantaminen

Reaktiivisessa funktio-ohjelmoinnissa on olemassa useita tapoja parantaa suorituskykyä. Tässä tutkimuksessa esitellään niistä kolme, jotka ovat signaalifunktioverkon optimointi, laiskan laskennan vähentäminen ja laskennan rinnakkaistaminen.

2.5.1 Signaalifunktioverkon optimointi

Signaalifunktioverkolla tarkoitetaan signaalifunktiota, joka koostuu monista yhteen liitetyistä signaalifunktioista, joiden keskinäiset syötteet ja ulostulot muodostavat riippuvuuksien verkon. Kun jonkin yksittäisen signaalifunktion ulostulo muuttuu,

niin muutos päivittyy koko signaalifunktioverkon läpi. Signaalifunktioverkon optimointia on tutkittu aikaisemmin, mutta tutkituissa optimointikeinoissa on usein rajoitteita, jotka estävät niiden käyttöä monimutkaisemmissa ohjelmissa.

Liu, Cheng ja Hudak (2009) ovat tutkineet kausaalisilla ja kommutatiivisilla nuolilla toteutetun reaktiivisen funktio-ohjelmoinnin optimointia. Tutkimuksen tavoitteena oli yksinkertaistaa signaalifunktioiden verkosto ohjelman käännön aikana. Yksinkertaistamisella saavutettiin huomattavia parannuksia suorituskykyyn, mutta optimointia pystyi hyödyntämään ainoastaan, jos optimoitava signaalifunktio pysyi rakenteeltaan aina samana.

Signaalifunktion rakenteen muuttumattomuus tarkoittaa sitä, että signaalifunktiot eivät pysty muuttamaan toimintaansa ohjelman suorituksen aikana. Tällöin edellä mainittu optimointitapa on käyttökelvoton peliohjelmoinnissa, sillä siinä on usein tarvetta rakenteelliselle dynaamisuudelle, josta on kerrottu luvussa 2.3.3.

Yksinkertaistamisen lisäksi muitakin optimointikeinoja on tutkittu. Esimerkiksi Sculthorpe ja Nilsson (2010) ovat esittäneet optimointikeinoja tarkkailemalla signaalifunktioverkon arvojen muutoksia, mutta heidän optimointikeinonsa ei toimi syklisille verkoille, eli verkoille, joissa on rekursiivisia määritelmiä, joille on usein tarvetta monimutkaisemmissa peleissä.

2.5.2 Laiskan laskennan vähentäminen

Laiska laskenta voi aiheuttaa joissain tapauksissa suorituskyvylle haittaa. Laskemattomille lausekkeille joudutaan varaamaan muistia ja ylimääräinen muistin varaaminen heikentää suorituskykyä (Ennals ja Jones 2003). Kääntäjä pystyy ohjelman staattisella analysoinnilla poistamaan laiskuutta paikoista, joissa sitä ei tarvita, mutta kääntäjän täytyy olla varovainen, jotta laiskaa laskentaa ei poisteta vääristä paikoista (Ennals ja Jones 2003).

GHC-kääntäjästä löytyy `Strict`- ja `StrictData` -kielilaajennukset, jotka poistavat laiskuuden kokonaan (GHC 2015). Näiden käyttö voi parantaa suorituskykyä, mutta ne aiheuttavat ongelmia paikoissa, joissa laiskuutta tarvitaan.

2.5.3 Rinnakkaisuus

Reaktiivisella funktio-ohjelmoinnilla tehtyjen ohjelmien rinnakkaistamista on tutkittu hyvin vähän. Peterson, Trifonov ja Serjantov (2000) yrittivät rinnakkaistaa reaktiivista funktio-ohjelmointia Linda-järjestelmässä. Kyseinen tutkimus havaintoineen oli Linda-järjestelmäkohtainen ja varhaisessa vaiheessa, joten sen tuloksia ei pysty soveltamaan tässä tutkimuksessa.

Rinnakkaisohjelmointi säikeillä ja lukoilla on perinteisesti ollut haasteellista yksinkertaisillekin ongelmille ja rinnakkaistettujen ohjelmien vikoja on ollut vaikea selvittää (Marlow 2013). Haskell-ohjelmointikieli kuitenkin tarjoaa paljon työkaluja rinnakkaislaskennan toteuttamiseen (Marlow 2013), joista osa on esitelty luvussa 2.4.

3 Tutkimuksen toteutus

Luvun alussa perehdytään tutkimusongelmaan, tutkimuksen tavoitteisiin ja käytettyyn metodiin. Tämän jälkeen esitellään tutkimuksessa toteutettu IT-artefakti, jonka avulla tarkastellaan rinnakkaisuuden vaikutusta suorituskykyyn testipelin avulla. Lisäksi pohditaan mittausmenetelmiä ja mittaukseen vaikuttavia parametreja.

3.1 Tutkimusongelma ja -kysymykset

Tämän tutkimuksen tutkimusongelmana on selvittää, miten suorituskykyä voidaan parantaa reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä.

Tutkimusongelma jakautuu kolmeen tutkimuskysymykseen:

1. Millaisia tapoja on parantaa suorituskykyä reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä?
2. Miten rinnakkaistamista tehdään reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä?
3. Miten rinnakkaistaminen vaikuttaa suorituskykyyn reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä?

Tutkimus rakentuu tutkimusongelman ja tutkimuskysymysten ympärille. Tavoitteena on tuottaa IT-artefakti, jonka tarkoituksena on parantaa luvussa 2.3 esiteltyä Netwire-nimistä reaktiivista funktio-ohjelmointikirjastoa lisäämällä sinne yleiskäyttöinen signaalifunktiokokoelman laskentaa rinnakkaistava funktio, joka parantaa reaktiivisella funktio-ohjelmoinnilla tehtävien pelien suorituskykyä.

3.2 Tutkimusmetodi

Tutkimusmetodina käytetään suunnittelutiedettä (design science). Hevner, March ja Park (2004) ovat määrittäneet seitsemän suuntaviivaa, joiden mukaan suunnittelutieteellinen tutkimus voidaan toteuttaa:

1. Artefaktin suunnittelu (Design as an Artifact)
2. Ongelman merkityksellisyys (Problem Relevance)
3. Artefaktin arviointi (Design Evaluation)
4. Tutkimuksen tuotokset (Research Contributions)
5. Tieteellinen tarkkuus (Research Rigor)
6. Suunnitteluprosessi etsintäprosessina (Design as a Search Process)
7. Tulosten kommunikointi (Communication of Research)

Ensimmäisen suuntaviivan mukaan suunnittelutieteellisen tutkimuksen tulee tuottaa merkityksellinen IT-artefakti (Hevner, March ja Park 2004). IT-artefaktilla tarkoitetaan tuloksena syntyviä tietotekniikkaan liittyviä käsitteitä, malleja, menetelmiä tai toteutuksia. Tässä tutkimuksessa pyritään löytämään signaalifunktiokokoelman laskennan rinnakkaistamiselle toteutus, jolla pystytään parantamaan reaktiivisella funktio-ohjelmoinnilla tehtävien pelien suorituskykyä.

Hevner, March ja Park (2004) ohjaavat toisessa suuntaviivassaan tarkastelemaan ratkaistavan ongelman merkityksellisyyttä sekä teknisestä näkökulmasta että liiketoimintaympäristön vaatimusten kannalta. Pelialalla kilpailu on kovaa ja erilaisten pelien tarjonta käyttäjille on laaja. Pelinkehittäjän näkökulmasta tämän tutkimuksen tuloksena syntyvä toteutus on merkityksellinen, koska sen avulla peleihin saa paremman suorituskyvyn. Käyttäjä kokee tämän ruudunpäivitysnopeuden kasvamisena, jolla voidaan taata sulava pelikokemus. Pelien kannalta ruudunpäivitysnopeuden olisi suotavaa olla minimissään 30 tai jopa 60 ruudunpäivitystä sekunnissa.

Reaktiivista funktio-ohjelmointia on käytetty myös kaupallisten pelien kehittämiseen (Maruseac 2017; Perez, Bärenz ja Nilsson 2016). Funktio-ohjelmoinnin yleistymisen myötä reaktiivisen funktio-ohjelmoinnin tutkiminen pelinkehittämisen näkökulmasta on merkityksellinen tutkimusaihe. Aiheesta on tehty vain vähän tutkimusta, eikä suorituskykyyn ole otettu kantaa aikaisemmin.

Rinnakkaistamisen valitsemista tutkimusongelman ratkaisemiseen tukee funktio-ohjelmoinnin valinta tutkimuksen lähtökohdaksi ja prosessorien ytimien määrän jatkuva kasvu kellotaajuuden kasvamisen sijaan. Rinnakkaisuus asettaa uusia haas-

teita, joiden ratkaisuun funktionaalinen ohjelmointiparadigma tuo helpotusta. Funktionaalinen ohjelmointi tapahtuu korkeammalla abstraktion tasolla imperatiiviseen ohjelmointiin verrattuna (Sutter ja Larus 2005). Funktio-ohjelmoijan ei usein tarvitse koskea synkronointiprimitiiveihin ja funktionaalisista ohjelmista löytyy luonnostaan monia helposti rinnakkaistettavia osia (Sutter ja Larus 2005). Prosessorin ytimien kellotaajuuden kasvattaminen tulee olemaan jatkuvasti haastavampaa virrankulutuksen ja prosessorin lämpenemisen takia, joten prosessorien valmistajat joutuvat kasvattamaan ytimien määrää tuodakseen prosessoreille lisää laskentatehoa (Herlihy ja Luchangco 2008; Sutter ja Larus 2005).

Kolmas suunnittelutieteen suuntaviiva tarkastelee syntyneen IT-artefaktin hyödyllisyyden, laadun ja vaikutuksen arviointia. Hevner, March ja Park (2004) edellyttävät, että artefaktin arvioinnissa määritetään oikeanlaiset mittarit ja mahdollisesti kerätään ja analysoidaan dataa. Heidän mukaansa artefaktia voidaan arvioida esimerkiksi toiminnallisuuden, suorituskyvyn tai luotettavuuden näkökulmasta. Arviointimenetelmänä tässä tutkimuksessa käytetään kokeellista menetelmää (engl. *controlled experiment*) (Hevner, March ja Park 2004). Tutkimusta varten luodaan peli, joka toimii kontrolloituna ympäristönä artefaktin vaikutusta mittattaessa.

Neljännen suuntaviivan mukaan tutkimuksen tulee tuottaa merkittävä artefakti, uutta tietoa tai uusia menetelmiä (Hevner, March ja Park 2004). Tämän tutkimuksen tuloksena syntyy artefakti, joka on yleisin tulos suunnittelutieteellisissä tutkimuksissa (Hevner, March ja Park 2004).

Viides suuntaviiva korostaa tieteellisen tarkkuuden ja täsmällisyyden merkityksellisyyttä. Suunnittelutietellinen tutkimus pohjautuu usein matemaattisiin malleihin, joilla kuvataan tuotettua artefaktia (Hevner, March ja Park 2004). Täsmällisyyteen päästään laadukkaalla tietämyskannalla, joka koostuu muun muassa viitekehyksestä, malleista ja metodeista (Hevner, March ja Park 2004). Tässä tutkimuksessa täsmällisyyteen pyritään luomalla viitekehys, joka koostuu mahdollisimman laadukkaasta lähdeaineistosta, sekä huolellisesti valituilla mittaustavoilla.

Suunnittelutieteen kuudennen suuntaviivan mukaan suunnitteluprosessia voidaan kuvata etsimisprosessina. Prosessissa tutkimusongelmaa pyritään yksinkertaistamaan ja tämän jälkeen siihen etsitään toimivin ratkaisu iteratiivisesti (Hevner, March ja Park 2004). Tätä tutkimusta tehtäessä pyrittiin etsimään erilaisia lähestymistapoja ongelman ratkaisemiseen, joita analysoimalla päädyttiin valitsemaan rinnakkaistaminen.

Viimeinen, seitsemäs suuntaviiva, ohjaa tutkijaa välittämään tulokset ymmärrettävässä muodossa sekä teknisesti orientoituneille, että liiketoimintasuuntautuneille henkilöille. Teknisesti orientoituneet henkilöt tarvitsevat riittävästi yksityiskohtia tuotetusta artefaktista sen jatkotutkimukselle ja toteuttamiseen liiketoimintaympäristössä. Liiketoimintasuuntautuneet henkilöt tarvitsevat riittävästi tietoa artefaktista päättääkseen kannattaako organisaation resursseja hyödyntää artefaktin toteuttamiseen heidän työympäristössään. (Hevner, March ja Park 2004.) Tässä tutkimuksessa tuotetun artefaktin toteuttamiseen on annettu riittävästi yksityiskohtia ja lisäksi tutkielma on pyritty kirjoittamaan niin, että siitä käy ilmi artefaktin merkitys, joka on myös liiketoimintasuuntautuneiden henkilöiden ymmärrettävissä.

3.3 Tutkimuksen toteutus

Tutkimus aloitettiin tutkimusongelman ja aiheen varmistuttua kesällä 2016. Tutkimusta varten tehtiin aluksi laaja kartoitus aiempaan tutkimukseen ja kirjallisuuteen. Kartoittamisen toteutus osoittautui haasteelliseksi, koska aiheesta on vain vähän aiempaa tutkimusta. Lähdeaineiston valinnassa noudatettiin periaatteita, joita ovat määrittäneet Hirsijärvi, Remes ja Sajavaara (2001). Valintaan vaikuttaneita tekijöitä olivat muun muassa kirjoittajan tunnettuus, lähdemateriaalin julkaisija ja lähteen uskottavuus. Lähdeaineisto valittiin siten, että se tukee tutkimuksessa tuotetun IT-artefaktin toimintaperiaatteen ymmärtämistä. Artefaktin toteuttamistavan valitsemisessa kiinnitettiin huomiota aiempiin tutkimuksiin ja niiden havaintoihin.

Lähdeaineiston kokoamisen jälkeen aloitettiin tutkimuksen seuraava vaihe, jossa valmistettiin testiympäristö tutkimuksessa tuotettavaa artefaktia varten. Artefakti

kehitettiin syklisesti useaan kertaan testaamalla ja mittaamalla. Artefaktin kehittämisen lisäksi tutkimusta tehtäessä kiinnitettiin erityistä huomiota mittausmenetelmään ja sen käyttämien parametrien valitsemiseen. Artefaktin vakiinnuttua erilaisia mittausmenetelmiä testattiin useaan otteeseen. Testaamisen päämääränä oli varmistua käytettävien menetelmien soveltuvuudesta kyseisen artefaktin tuottaman parannuksen mittaamiseen. Menetelmän valinnassa pyrittiin varmistumaan siitä, että valittu mittaustapa mittaa artefaktin toimintaa oikein ja luotettavasti.

Tutkimusta suunniteltaessa testattiin laiskan laskennan vähentämisen vaikutuksia testipelin suorituskykyyn. Alustavien mittausten perusteella havaittiin parannuksen olevan huomattavasti rinnakkaistamisella saavutettua parannusta heikompi, joten laiskan laskennan vähentäminen jätettiin pois tästä tutkimuksesta. Haskell-ohjelmointikielen tarjoamat monipuoliset toteutustavat rinnakkaistamiseen ja tutkimuksen suunnitteluvaiheen alustavat mittaukset tekivät rinnakkaisuudesta luontevan lähestymistavan tämän tutkimuksen tutkimusongelman ratkaisemiseen.

3.4 Rinnakkaistamisen toteuttaminen

Netwire-kirjaston signaalifunktiot voivat olla sivuvaikutuksellisia. Tämä tuottaa ongelmia rinnakkaisuuden kanssa. Sivuvaikutusten suoritusjärjestyksellä on merkitystä, mutta rinnakkaistettuna signaalifunktioiden arvot lasketaan ennalta arvaamattomassa järjestyksessä. Sivuvaikutuksia voi rajoittaa ottamalla `Wire`-tyypin sijaan sivuvaikutuksettoman `WireP`-tyypin käyttöön. Listauksessa 34 on esitetty luvussa 2.3.5 olevasta `stepWire`-funktiosta versio, jossa on otettu käyttöön `WireP`-tyyppi `Wire`-tyypin tilalle.

Listaus 34. Signaalifunktiokokoelman evaluaatio `WireP`-tyypillä

```
stepWires :: Monoid s
          => WireP s e ([WireP s e i b], i) [(b, WireP s e i b)]
stepWires =
    mkGen $ \dt (wires, inp) → do
        let step w = runIdentity $ stepWire w dt (Right inp)
        let output = map step wires
        let filteredOutput = [(o, w') | (Right o, w') ← output]
        return (Right filteredOutput, stepWires)
```

Signaalifunktiolistan laskennan voi rinnakkaistaa luvussa 2.4 esitetyillä funktioilla ja menetelmillä. Listauksessa 35 on esitetty toteutus funktiosta, joka toimii `stepWires`-funktion tavoin, mutta laskenta tapahtuu rinnakkaisesti. Suurimmat eroavaisuudet `stepWires`-funktioon on merkattu vihreällä taustavärillä. Jokaiselle signaalifunktiolle kutsutaan `step`-apufunktiota, jonka lasku rinnakkaistetaan luvussa 2.4.3 esitetyllä `parList`-funktiolla. Rinnakkaistamisessa on riskinä se, että laiskan laskennan seurauksena rinnakkaisesti luodaan vain kesken jääneitä laskemattomia lausekkeita. `step`-funktion kutsut suoritetaan rinnakkain ja sen sisällä oleva `deepseq`-funktio varmistaa, että signaalifunktion arvon lasku suoritetaan loppuun asti, eli *normal form* -muotoon. `deepseq`-funktion käyttö vaatii `e`- ja `b`-tyyppimuuttujille `NFData`-tyyppiluokkarajoitukset. *Normal form* -muotoon laskenta on tehtävissä myös vaihtamalla `rseq`-evaluointistrategian tilalle `rdeepseq`.

Listaus 35. Signaalifunktiokokoelman rinnakkainen evaluaatio

```
parStepWires :: (NFData e, NFData b, Monoid s)
             => WireP s e ([WireP s e i b], i) [(b, WireP s e i b)]
parStepWires =
    mkGen $ \dt (wires, inp) → do
        let step w = runIdentity $ do
                (v, w') ← stepWire w dt (Right inp)
                return $ v `deepseq` (v, w')
        let output = map step wires `using` parList rseq
        let filteredOutput = [(o, w') | (Right o, w') ← output]
        return (Right filteredOutput, parStepWires)
```

Rinnakkaistamiseen voi soveltaa luvussa 2.4.4 esitettyä lohkotusta, jolla voi olla parantavia vaikutuksia suorituskykyyn. Listauksessa 36 on esitetty `parStepWires`-funktiosta versio, jossa käytetään lohkotusta `parListChunk`-funktion avulla. Funktioon on lisätty parametriksi lohkon koko. Erot `parStepWires`-funktioon on merkattu vihreällä taustavärillä.

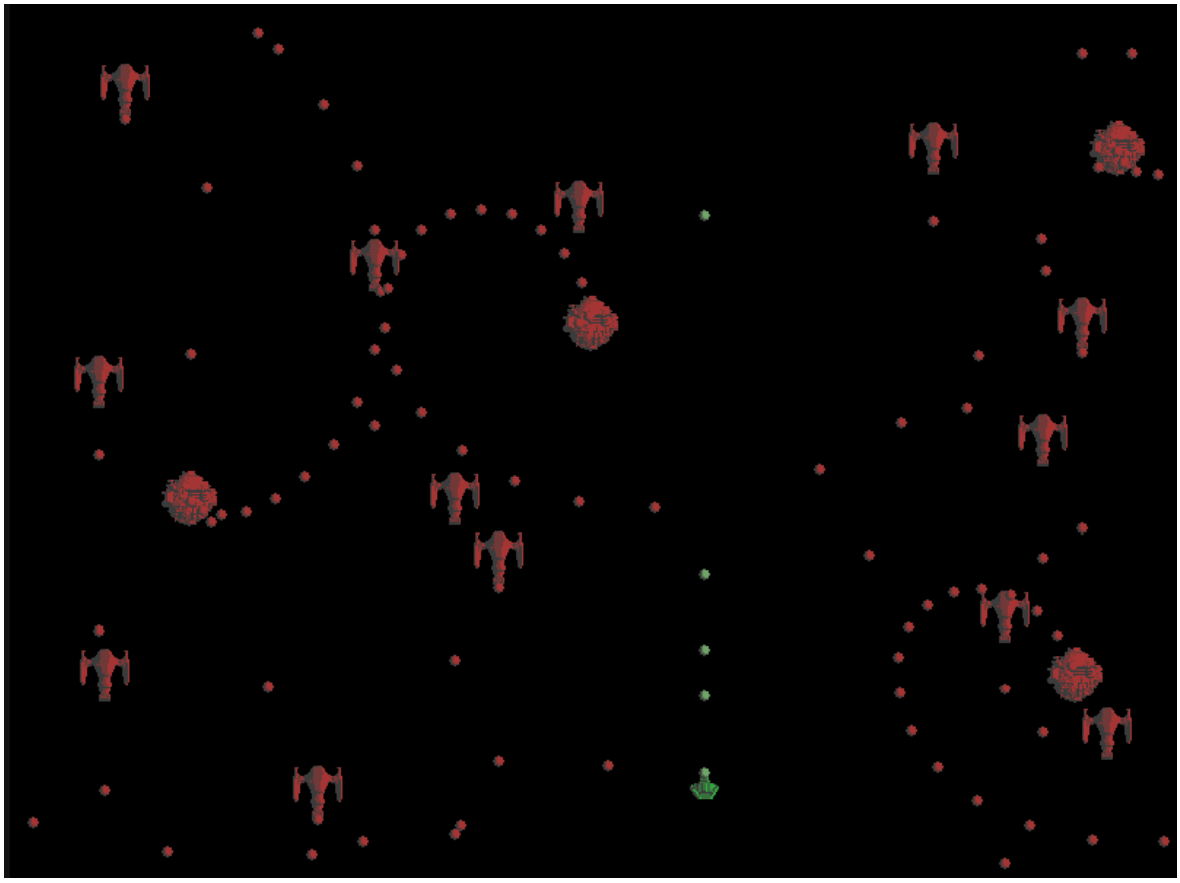
Listaus 36. Signaalifunktiokokoelman rinnakkainen evaluaatio lohkotuksella

```
parStepWiresChunk :: (NFData e, NFData b, Monoid s)
                  => Int
                  -> WireP s e ([WireP s e i b], i) [(b, WireP s e i b)]
parStepWiresChunk chunkSize =
    mkGen $ \dt (wires, inp) -> do
        let step w = runIdentity $ do
                (v, w') <- stepWire w dt (Right inp)
                return $ v `deepseq` (v, w')
        let output = map step wires `using` parListChunk chunkSize rseq
        let filteredOutput = [(o, w') | (Right o, w') <- output]
        return (Right filteredOutput, parStepWiresChunk chunkSize)
```

3.5 Testiympäristön esittely

Suorituskykymittausta varten kehitettiin testiympäristö, jossa mitattiin tutkimuksessa toteutetun IT-artefaktin suorituskykyä. Testiympäristö on reaktiivisella funktio-ohjelmoinnilla ja Netwire-kirjastolla toteutettu peli. Pelissä pelaaja ohjaa näytön alareunassa olevaa avaruusalusta, joka ampuu ylhäältä ilmestyviä vihollisaluksia. Pelaajan ammukset tuhoavat vihollisaluksia osuttuaan niihin riittävän monta kertaa. Vihollisaluksia ilmestyy jatkuvasti lisää ja ne ampuvat tietyn ajan välein ammuksia, joita pelaajan aluksella väistellään. Pelissä on kahdenlaisia vihollisaluksia. Ne ampuvat ammuksia eri ampumistiheyksillä ja kestävät eri määrän pelaajan ammuksien osumia ennen tuhoutumistaan. Esimerkki pelitilanteesta on esitetty kuviossa 6.

Peli on toteutettu siten, että siinä on mahdollista lisätä laskennallista kuormaa nostamalla vihollisaluksien ilmestymistiheyttä ja ampumistiheyttä, jonka myötä peliobjektien lukumäärä kasvaa. Tämän kuormittamisen avulla pystytään mittaamaan



Kuvio 6. Kuvankaappaus suorituskykymittauksessa käytetystä pelistä.

tutkimuksessa toteutetun IT-artefaktin vaikutusta. Mittauksen helpottamiseksi pelilogiikan laskenta tapahtuu täysin deterministisesti, jolloin samaa ruudunpäivitystä useaan kertaan mitatessa mittaustuloksiin vaikuttavat vain pelin ulkopuoliset tekijät.

Pelin piirtäminen on toteutettu SDL2-kirjastolla. Suorituskykymittausten ajan pelin piirtäminen ei ollut käytössä, jotta mittaus kohdistui pelkästään peliobjektien päivittämiseen.

3.6 Mittausmenetelmä

Ohjelman suorituskykyyn vaikuttaa mittausympäristö. Pienet, näennäisesti merkityksettömän oloiset tekijät voivat vaikuttaa suorituskykyyn (Mytkowicz ym. 2009), jolloin yksittäiseen mittaukseen voi tulla mittavirheitä. Esimerkiksi taustalla toimivat

käyttöjärjestelmän muut prosessit voivat aiheuttaa virhettä tuloksiin kilpailemalla prosessorin suoritusajasta (Mytkowicz ym. 2009). Myös sillä, millaista prosessoria käytetään, on vaikutusta mittauksen tuloksiin. Sama ohjelma voi antaa suorituskykymittauksissa erilaisia tuloksia riippuen prosessorin arkkitehtuurista (Hazelhurst 2010). Prosessorin kellotaajuus vaikuttaa myös mittaustuloksiin, mutta prosessorin L2-välimuistin määrällä on kellotaajuutta enemmän vaikutusta (Hazelhurst 2010).

Monet modernit kielet, kuten Java, C# ja Ruby, tarjoavat turvaa muistin hallintaan ja automaattisen roskienkeruun, jota esimerkiksi C:ssä ja C++:ssa ei ole (Blackburn ym. 2008). Haskell-ohjelmointikielestä löytyy tehokas roskienkeruun algoritmi, joka on optimoitu siihen, että uusia objekteja luodaan ja tuhotaan valtavia määriä jatkuvasti (Marlow ym. 2008). Roskienkeruu aiheuttaa kuitenkin haasteita suorituskyvyn mittaukselle, koska se tapahtuu satunnaisin väliajoin ja ohjelman suoritus pysähtyy roskienkeruun ajaksi. Pinomuistin määrä vaikuttaa myös roskienkeruuseen ja sitä kautta myös suorituskykyyn (Blackburn ym. 2008). Roskienkeruusta ja siihen liittyvien parametrien määrityksestä kerrotaan lisää luvussa 3.7.

Kalibera ja Jones (2013) ovat havainneet, että tutkimuksissa, joissa parannus on huomattavan suuri, ei ole käytetty äärimmäisen tarkkoja mittausmenetelmiä. Tarkkoja menetelmiä tulisi soveltaa tilanteissa, joissa saavutettu parannus jäisi lähelle 10% tasoa, jolloin tulokset saattaisivat selittyä mittausten epätarkkuudella tai mittausvirheillä. Tässä tutkimuksessa parannus ylittää 10% tason.

Tutkimuksessa toteutetussa suorituskykymittauksessa selvitetiin miten signaalifunktioiden laskennan rinnakkaistaminen vaikuttaa testipelin suorituskykyyn. Pelissä olemassaolevia objekteja kuvastetaan signaalifunktioilla. Suorituskyvyn näkökulmasta on olennaista tällöin mitata, paljonko aikaa kuluu signaalifunktioiden arvojen laskemiseen. Suoritukykymittauksilla mitatuista ajoista laskettiin keskiarvot, joiden perusteella piirrettiin kuvaajat.

Pelin suorituskykyä mitattiin kasvattamalla objektien määrää pelin edetessä 2000 objektiin, joka kuormittaa suorituskykyä. Mittaustuloksissa olevat mittapisteet muodostuivat mittaamalla pelitilanteen päivitykseen käytettyä aikaa suhteessa objektien

määrään kyseisellä hetkellä.

Jokainen mitattu pelitilanteen päivitys laskettiin 500 kertaa peräkkäin. Toistoilla pyrittiin vähentämään mittaustilanteen virheitä. Esimerkiksi pienillä, alle sadan objektin määrillä, pelitilanteen päivitykseen käytetty aika oli testiympäristössä niin pieni, ettei sitä ollut mahdollista mitata tarkasti. Toistojen määrä on suuri myös sen takia, että mittausten tulokset vaihtelevat hieman ohjelmien suorittamisen epädeterministisestä luonteesta johtuen (Kalibera ja Jones 2013). Epädeterminismiä pyrittiin testipelin tapauksessa lieventämään sillä, että pelilogiikan laskeminen suoritettiin täysin deterministisesti.

Suorituskykyyn vaikuttaa huomattavasti asetukset, joilla testattavan ohjelma käännetään (Hazelhurst 2010). Mittauksissa käytettiin joko `stepWires`-funktiota tai `parStepWires`-funktiota eri säikeiden määrillä ja kääntämisen asetuksiin kiinnitettiin erityistä huomiota. Rinnakkaistumattoman version suorituskyky mitattiin, jotta voitiin vertailla kuinka suuren parannuksen rinnakkaisuus tuotti. Rinnakkaistumaton versio käytti `stepWires`-funktiota ja se käännettiin GHC vivuilla `-O2` ja `-threaded`. Vivulla `-O2` maksimoitiin koodin optimointi ja vivulla `-threaded` sallittiin rinnakkaisuus, jota `stepWires`-funktio ei hyödynnä, mutta sillä sallittiin ohjelman käyttää GHC-kääntäjän rinnakkaistuvaa roskienkeruuta, jotta rinnakkaistuva versio ei saa ylimääräistä parannusta suorituskykyyn siitä. Rinnakkaistuva versio käytti `parStepWires`-funktiota ja se käännettiin GHC vivuilla `-O2`, `-threaded` ja `-rtsopts`, joista viimeinen mahdollistaa säikeiden määrän rajoittamisen ennen ohjelman käynnistystä.

Ajan mittaaminen toteutettiin clock-kirjastolla, joka tarjoaa useita keinoja tietokoneen ajan kysymiseen (Sert 2016b). Parhaat vaihtoehdot olivat `Realtime`- ja `Monotonic`-tyyppiset ajan mittaamiset. `Realtime`-tyyppinen ajan mittaaminen käyttää Windows käyttöjärjestelmässä Windows API:n `GetSystemTimeAsFileTime`-funktiota (Sert 2016a), joka antaa tietokoneen kellonajan 100 nanosekunnin tarkkuudella (Microsoft 2017b), mutta alustavien mittausten perusteella tällä tavalla saatu aika voi poiketa oikeasta ajasta jopa 15 millisekunnin verran. Tämän keinon tarkkuus on siis hyvä, mutta tuloksen virheellisyys voi olla suuri.

`Monotonic`-tyyppinen ajan mittaaminen käyttää ajan saamiseen `QueryPerformanceCounter`- ja `QueryPerformanceFrequency`-funktioita. Näiden avulla saadun ajan absoluuttisella arvolla ei ole mitään järkevää merkitystä, mutta kahden mitatun ajan erotus on lähempänä oikeasti kulunutta aikaa, kuin `Realtime`-tyyppisellä mittaustavalla mitattuna (Microsoft 2017a). Ajan mittaamiseen valittiin `Monotonic`-tyyppinen mittaustapa paremman tarkkuuden takia.

3.7 Mittauslaitteisto ja parametrien määritys

Suorituskykymittauksiin vaikuttaa mittauksessa käytetyn tietokoneen prosessori, säikeiden määrä ja roskienkeruu. Listauksessa 36 esitetyn `parStepWiresChunk`-funktion tapauksessa suorituskykyyn vaikuttaa myös valittu lohkon koko.

Suorituskykymittauksessa käytetyssä tietokoneessa oli 64-bittinen versio Windows 10 Education -käyttöjärjestelmästä ja Intel Core i7-3770K prosessori, jossa on neljä ydintä, jotka voivat suorittaa yhtäaikaisesti kahdeksaa säiettä. Prosessorin kellotaajuus oli rasituksen alaisena 3.9 Ghz. Prosessorin L3-välimuistin määrä on 8 megatavua, joka on jaettu kaikkien ytimien kesken (Intel 2017). Jokaisella ytimellä on oma 0.25 megatavun L2-välimuisti, jonka määrä varmistettiin CPU-Z ohjelmalla (CPUID 2017). GHC:n versio oli 8.0.1 ja olennaisimmat moduulit ja niiden versiot olivat `netwire-5.0.2` ja `clock-0.7.2`.

Kuten luvussa 3.6 todettiin, suorituskykyyn vaikuttaa roskienkeruu. Haskell-ohjelmissa tiedon muuttumattomuus pakottaa luomaan runsaat määrät tilapäistä tietoa (roskaa), mutta tiedon muuttumattomuus sallii myös tehokkaan roskienkeruun (GHC 2014). Haskell-ohjelmat saattavat helposti luoda tietoa gigatavun verran sekunnissa (GHC 2014).

GHC-kääntäjän roskienkeruussa käytetty algoritmi on nimeltään *parallel generational-copying garbage collection*, jonka toiminnasta ja rinnakkaistamisesta Marlow ym. (2008) ovat kirjoittaneet tarkemmin. Muisti on jaettu neljän kilotavun lohkoihin ja lohkot on jaettu eri generaatioihin (Marlow, Peyton Jones ja Singh 2009). Generaatioihin jakaminen on saanut alkunsa ideasta, jonka on alunperin esittänyt Ungar (1984). Uusi

data luodaan erityiselle allokaatioalueelle. Kun allokaatioalue täyttyy, niin kaikki säikeet lopettavat laskennan roskienkeruun ajaksi (Marlow, Peyton Jones ja Singh 2009), joka voidaan suorittaa rinnakkaisesti (Marlow ym. 2008). Allokaatioalueen täytyttyä arvot, jotka eivät ole roskaa, kopioidaan ensimmäiseen generaatioon talteen, jolloin roska-arvoihin ei kosketa. Tämän seurauksena mitä suurempi osa tuotetuista arvoista on roskaa, sitä nopeammin roskienkeruu toimii (GHC 2014). Mittauksen kannalta sopivimman allokaatioalueen koko määritetään luvussa 3.7.1. Generaatiot sijaitsevat pinomuistissa ja GHC-kääntäjälle voi antaa suosituksen pinomuistin määrästä. Sopiva pinomuistin suosituksen määrä määritetään luvussa 3.7.2.

3.7.1 Allokaatioalueen koko

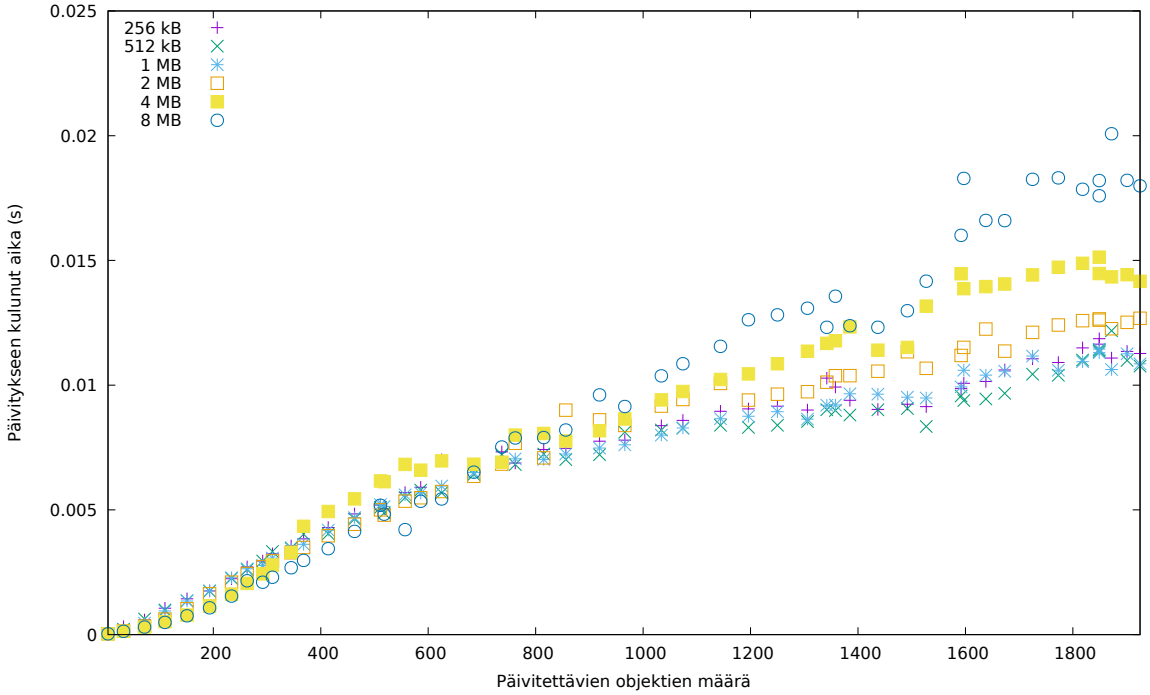
Marlow (2013) on ohjeistanut, että allokaatioalueen koko on suositeltavaa asettaa suurinpiirtein samaksi kuin yksittäisen prosessorin ytimen L2-välimuistin määrä. L2-välimuistin määrät vaihtelevat prosessoreittain. Prosessoreilla voi lisäksi olla L3-valimuistia ja välimuisti voi joskus olla jaettu kaikkien ytimien kesken, joten sopivalle allokaatioalueen koolle on hankala antaa tarkkaa optimaalista kokoa (Marlow 2013).

Allokaatioalueen oletuskoko on 512 kilotavua (GHC Team 2015). RTS parametrilla `-A` voidaan määrittää allokaatioalueen koko. Sopiva allokaatioalueen koko valittiin mittaamalla allokaatioalueen koon vaikutus suorituskykyyn. Mittauksen tulokset on esitetty kuviossa 7.

Kuvion 7 mittapisteet kuvastavat päivitykseen käytettyä aikaa eri kokoisella allokaatioalueella. Kuvion tuloksista voidaan havaita, että suorituskyky heikkenee allokaatioalueen koon kasvaessa. Allokaatioalueen kooksi valittiin oletusmäärä, eli 512 kilotavua.

3.7.2 Pinomuistin suositeltu määrä roskienkeruulle

RTS parametri `-H` määrittää roskienkerääjälle tietyn suositusmäärän verran pinomuistia käyttöön (GHC Team 2015). Suositusmäärä ei rajaa pinomuistin maksimimäärää. Oletusmäärä on pinomuistin suositukselle on 0 tavua, joka tarkoittaa poikkeukselli-



Kuvio 7. Allokaatioalueen koon vaikutus suorituskykyyn.

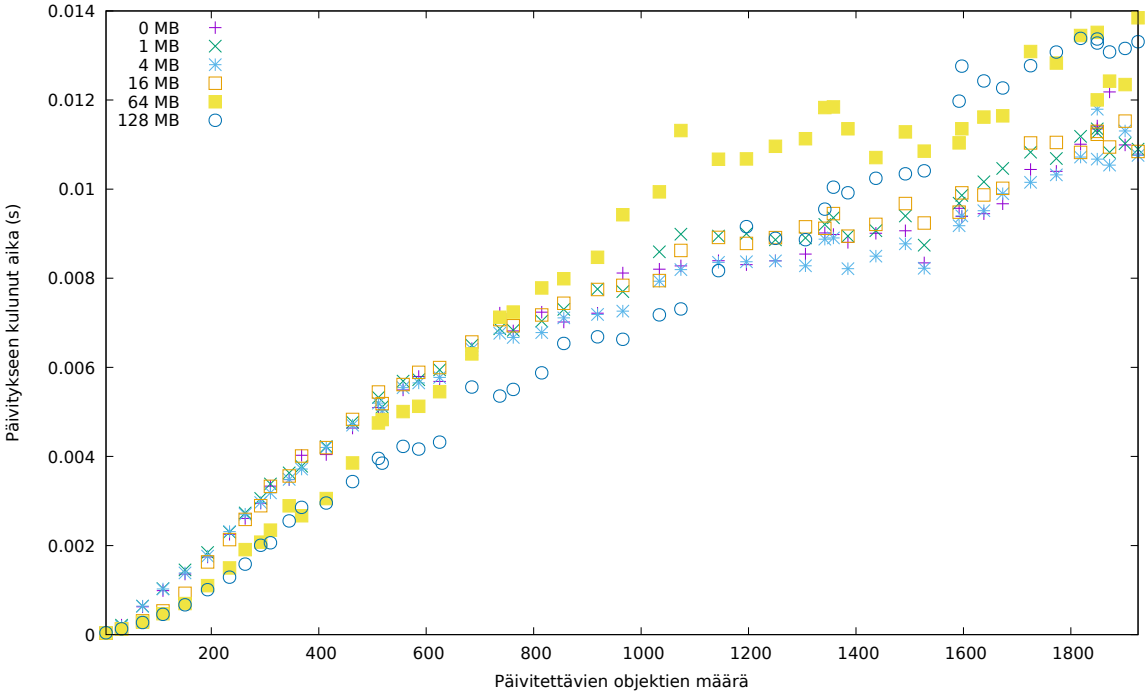
sesti sitä, että roskienkerääjä käyttää pinon kokona edellisen roskienkeruun aikana ollutta pinon kokoa (GHC Team 2015).

Sopiva pinomuistin suositusmäärä valittiin mittaamalla suositusmäärän vaikutusta suorituskykyyn. Mittauksen tulokset on esitetty kuviossa 8.

Kuvion 8 mittapisteet kuvastavat päivitykseen käytettyä aikaa eri kokoisella pinomuistin suositusmäärällä. Kuvion tuloksista voidaan havaita, että suorituskyky heikkenee suosituskoon kasvaessa. Pinomuistin suosituskooksi valittiin oletusmäärä, eli 0 tavua, jolloin pinomuistin määrä toimii edellä mainitun poikkeussäännön mukaisesti.

3.7.3 Lohkon koko

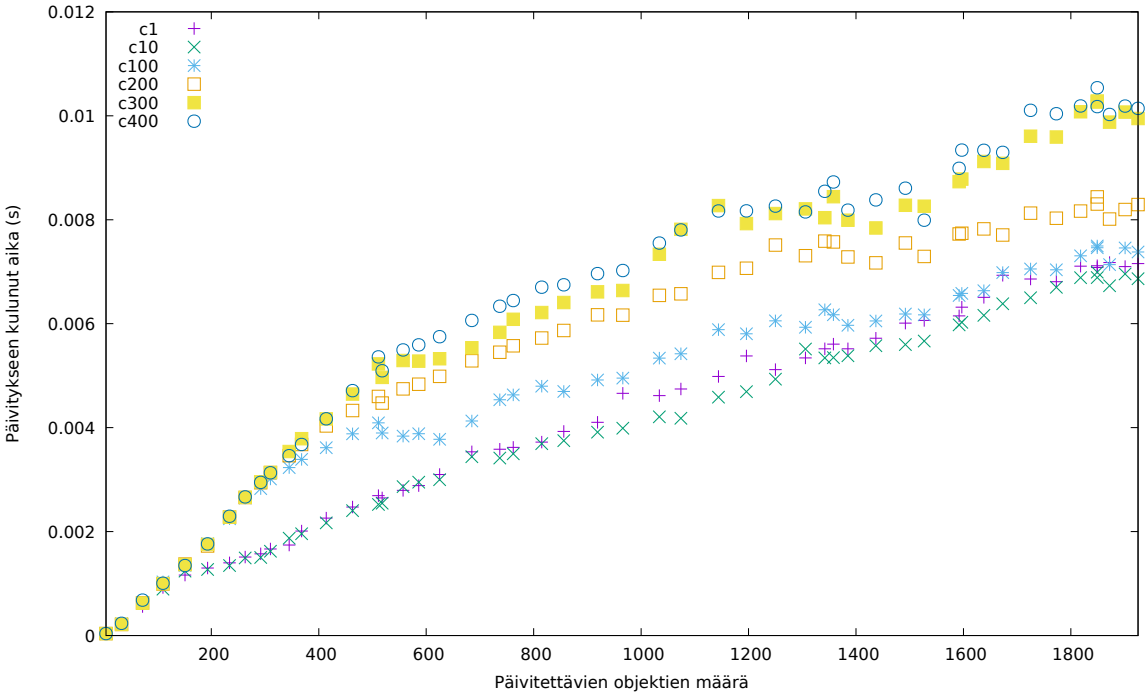
Listauksessa 36 esitetylle funktiolle `parStepWiresChunk` määritettiin optimaalinen lohkon koko. Sopivan lohkon koon voisi intuitiivisesti ajatella olevan sellainen, jolla kaikki objektit saataisiin jaoteltua muutamaan lohkoon, jotta jokainen säie voi



Kuvio 8. Pinomuistin suositusmäärän vaikutus suorituskykyyn.

laskea yhden lohkon objekteja. Esimerkiksi 2000 objektilla tämä voisi olla 200 – 500 objektia jokaista lohkoa kohden, jolloin lohkoja tulisi 4 – 10 kappaletta. Sopiva lohkon koko varmistettiin mittaamalla lohkon koon vaikutusta suorituskykyyn. Mittauksen tulokset on esitetty kuviossa 9. Kuviossa on esitetty mittaustulokset neljällä säikeellä. Mittaustulokset olivat vastaavat myös muillakin säiemäärillä.

Kuvion 9 mittapisteet `c1` – `c400` kuvastavat päivitykseen käytettyä aikaa eri kokoisilla lohkoilla. Esimerkiksi `c200` tarkoittaa, että kaikki objektit jaettiin 200 objektin lohkoihin. Toisin kuin voisi olettaa, kuvion tuloksista voidaan havaita, että lohkon koon tulee tässä tapauksessa olla pieni. Pienemmät lohkojen koot antoivat paremman suorituskyvyn myös suuremmilla objektimäärillä ja suorituskyky heikkeni lohkon koon kasvaessa. Suorituskykymittaukseen `parStepWiresChunk`-funktiolle valittiin lohkon kooksi 10 objektia.



Kuvio 9. Lohkon koon vaikutus suorituskykyyn.

4 Tulokset

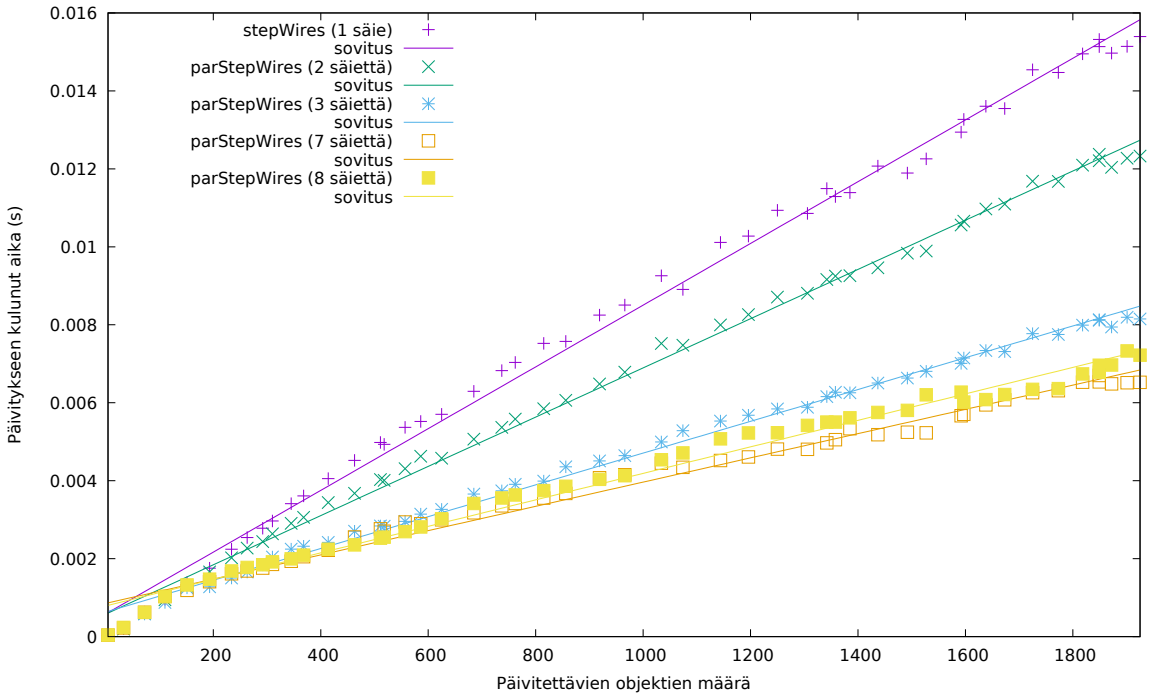
Luvussa esitellään tutkimuksessa toteutetulle IT-artefaktille tehtyjen suorituskykymittausten tulokset. Suorituskykymittaukset tehtiin testiympäristössä kolmella eri signaalifunktiokokoelman päivitysfunktiolla. Mittaukset tehtiin luvussa 3.4 esitetyillä funktioilla `parStepWires` ja `parStepWiresChunk`, jotka rinnakkaistavat laskentaa, ja vertailun vuoksi luvussa 2.3.5 esitetyllä funktiollä `stepWires`, jonka laskenta suoritetaan sarjassa.

`parStepWires`-funktion mittauksen tulokset on esitetty luvussa 4.1 ja `parStepWiresChunk`-funktion tulokset luvussa 4.2. Lopuksi luvussa 4.3 tutkitaan IT-artefaktin ja prosessorin ytimien määrän vaikutusta koko pelin suorituskykyyn.

4.1 Rinnakkaistamisen vaikutus suorituskykyyn

Kuviossa 10 on havainnollistettu mittapisteitä ja suoran sovituksia peliin tehdylle suorituskykymittaukselle, jossa käytettiin `parStepWires`-funktiota. Kuvion 10 suorien sovitukset tehtiin Marquardt-Levenberg algoritmilla. Suoran sovitusta vastaa lineaarinen funktio $f(x) = ax + b$, missä a on kulmakerroin ja b on vakio. Mittapisteet ”stepWires (1 säie)” kuvaavat mittausta, jossa käytettiin `stepWires`-funktiota, joka ei rinnakkaistu usealle ytimelle. Kuvion 10 muissa mittapisteissä mittauksessa käytettiin `parStepWires`-funktiota ja käytettävissä olevien säikeiden määrä oli 2, 3, 7 tai 8. Kokonaisuutena mittaus suoritettiin jokaisella säiemäärällä kahdesta kahdeksaan. Kuviosta 10 jätettiin selkeyden vuoksi pois säiemäärien 4 – 6 suorat ja mittapisteet. Nämä olisivat sijoittuneet kuviossa säiemääriä 3 ja 8 kuvaavien suorien väliin.

Taulukossa 2 on esitetty kulmakertoimet mittauksille, joissa käytettiin `parStepWires`-funktiota ja säikeiden määriä 2 – 8. Kulmakertoimet kuvastavat yksittäisen objektin päivittämiseen käytettyä aikaa. Taulukkoon on lisätty sovituksen kulmakertoimen virheet ja virheen suhteellinen osuus kulmakertoimesta. Taulukosta voidaan havaita, että säikeiden määrän kasvattaminen pienentää kulmakerrointa, jolloin yksittäisen objektin päivittämiseen käytetään vähemmän aikaa.



Kuvio 10. Mittapisteet ja suoran sovitukset suorituskykymittauksille.

Vertailun vuoksi mittaus suoritettiin myös `stepWires` funktiolla, joka ei rinnakkaista laskentaa useille ytimille. Suoran sovituksessa mittapisteille ”stepWires (1 ydin)” sovitetun suoran kulmakertoimeksi saatiin $7.919 \cdot 10^{-6}$ sekuntia/objekti. Sovituksessa kulmakertoimen virheeksi saatiin $7.788 \cdot 10^{-8}$ (0.9835%).

Kuviosta 10 ja taulukosta 2 voidaan havaita, että seitsemällä säikeellä saavutettiin parempi suorituskyky kuin kahdeksalla säikeellä. Tämä johtuu siitä, että optimaalinen säikeiden määrä ei ole aina sama kuin suurin mahdollinen samanaikaisesti suoritettavien säikeiden määrä, koska tietokoneen muut prosessit kilpailevat samanaikaisesti suoritusajasta (Marlow 2013).

Mittaukseen käytetyssä prosessorissa oli neljä ydintä, joista jokainen voi suorittaa kahta säiettä samanaikaisesti. Tämä ominaisuutta kutsutaan nimellä *hyper-threading* (Intel 2017). Säiemäärät 1 – 4 pystytään ajamaan itsenäisillä ytimillä, mutta säiemäärillä 5 – 8 hyper-threading tulee käyttöön, joka parantaa suorituskykyä, mutta ei anna yhtä suurta parannusta kuin mitä oikea ydin antaisi (Marlow 2013). Taulukosta 2 voi havaita, että suorituskyky parantuu vain marginaalisesti neljännen säikeen jäl-

Taulukko 2. Kulmakertoimet `parStepWires`-funktiolle

Säikeiden määrä	Kulmakerroin	Kulmakertoimen virhe ja suhteellinen osuus
2	$6.31508 \cdot 10^{-6}$ sekuntia/objekti	$\pm 5.368 \cdot 10^{-8}$ (0.8501%)
3	$4.07673 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.497 \cdot 10^{-8}$ (1.103%)
4	$3.46273 \cdot 10^{-6}$ sekuntia/objekti	$\pm 8.871 \cdot 10^{-8}$ (2.562%)
5	$3.18077 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.829 \cdot 10^{-8}$ (1.518%)
6	$3.21495 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.74 \cdot 10^{-8}$ (1.474%)
7	$3.11021 \cdot 10^{-6}$ sekuntia/objekti	$\pm 5.872 \cdot 10^{-8}$ (1.888%)
8	$3.3986 \cdot 10^{-6}$ sekuntia/objekti	$\pm 5.71 \cdot 10^{-8}$ (1.68%)

keen. Marlow (2013) on ehdottanut, että sopiva määrä säikeille on sama kuin ytimien määrä, jolloin taustalla olevat prosessit eivät häiritse ohjelman suoritusta liikaa.

4.2 Lohkotuksen vaikutus suorituskykyyn

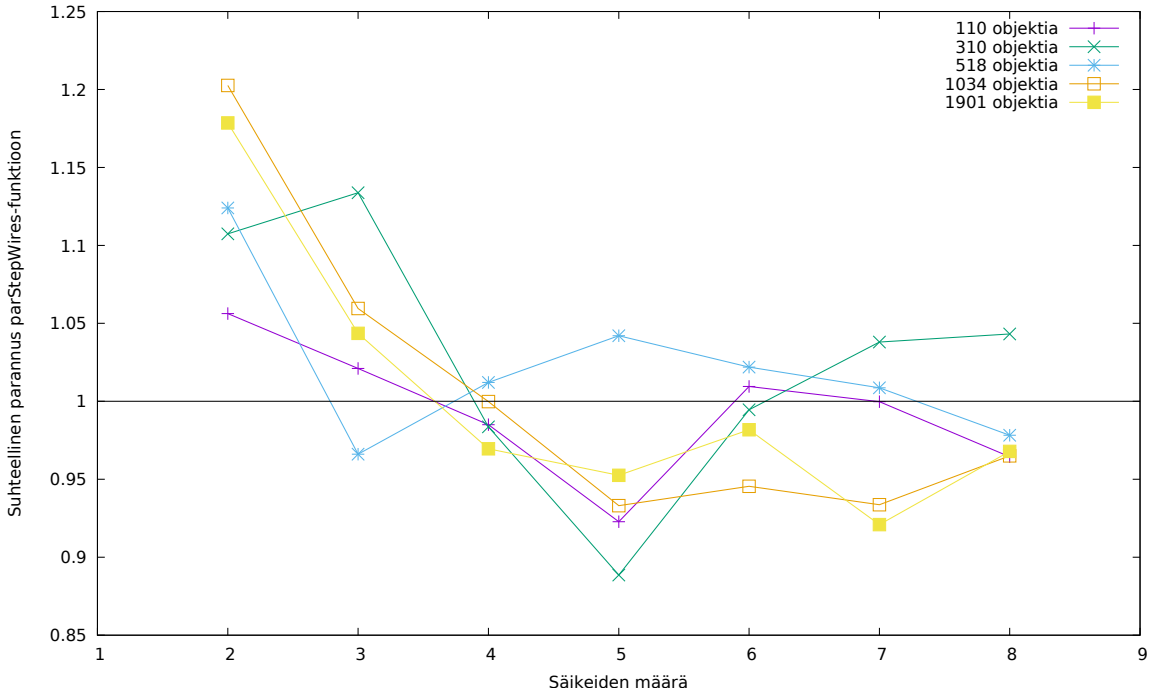
Suorituskykymittaukset tehtiin myös käyttäen `parStepWiresChunk`-funktiota, joka käyttää lohkottamista paremman suorituskyvyn saavuttamiseksi. Mittaus suoritettiin samaan tapaan kuin `parStepWires` funktion mittaus. Lohkon kooksi valittiin luvussa 3.7.3 tehtyjen havaintojen pohjalta 10 objektia.

Taulukossa 3 on esitetty kulmakertoimet mittauksille, joissa käytettiin `parStepWiresChunk`-funktiota ja säikeiden määriä 2 – 8. Kulmakertoimet kuvastavat yksittäisen objektin päivittämiseen käytettyä aikaa. Taulukkoon on lisätty sovituksen kulmakertoimen virheet ja virheen suhteellinen osuus kulmakertoimesta.

Kuten taulukosta 3 voidaan havaita, lohkotuksen kanssa tulokset olivat samankaltaisia `parStepWires`-funktion kanssa tehtyn mittauksen kanssa, mutta suorituskyky vaihtelee hieman riippuen säikeiden määrästä. Kuviossa 11 on esitetty lohkottamisen suhteellinen parannus `parStepWires`-funktioon verrattuna. Pisteiden sijainti on saatu jakamalla `parStepWires`-funktiolla objektien päivitykseen käytetty aika `parStepWiresChunk`-funktiolla käytettyyn aikaan.

Taulukko 3. Kulmakertoimet `parStepWiresChunk`-funktiolle

Säikeiden määrä	Kulmakerroin	Kulmakertoimen virhe ja suhteellinen osuus
2	$5.29479 \cdot 10^{-6}$ sekuntia/objekti	$\pm 5.061 \cdot 10^{-8}$ (0.9558%)
3	$3.88427 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.717 \cdot 10^{-8}$ (1.214%)
4	$3.36466 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.41 \cdot 10^{-8}$ (1.311%)
5	$3.35798 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.766 \cdot 10^{-8}$ (1.419%)
6	$3.35566 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.695 \cdot 10^{-8}$ (1.399%)
7	$3.47187 \cdot 10^{-6}$ sekuntia/objekti	$\pm 5.417 \cdot 10^{-8}$ (1.56%)
8	$3.74311 \cdot 10^{-6}$ sekuntia/objekti	$\pm 4.795 \cdot 10^{-8}$ (1.281%)



Kuvio 11. Lohkottamisen suhteellinen suorituskyvyn parannus `parStepWires`-funktioon verrattuna.

Lohkottaminen ei tuottanut tässä tapauksessa automaattisesti parempaa suorituskykyä rinnakkaistuvaan lohkottamattomaan `parStepWires`-funktioon verrattuna. Lohkotettuna suorituskyky oli kuitenkin parempi rinnakkaistumattomaan `stepWires`-funktioon verrattuna, mutta optimaalisen suorituskyvyn saavuttaminen vaatii

tarkkaa tasapainottelua säikeiden määrän ja lohkon koon välillä.

4.3 IT-artefaktin ja ytimien määrän vaikutus koko pelin suorituskykyyn

Pelilogiikan laskennan rinnakkaistamisen vaikutusta koko testipelin toimintaan tutkittiin ThreadScopen (Jones, Marlow ja Singh 2009) avulla. Peliobjekteja päivitettiin `stepWires-` tai `parStepWires`-funktiolla ja ne piirrettiin näytölle SDL2-kirjaston avulla. Mitatun ruudunpäivityksen hetkellä peliobjekteja oli 1912 kappaletta. HEC-laskentakonteksteja oli neljä kappaletta, joiden avulla prosessorin ytimet pystyivät suorittamaan laskentaa rinnakkain.

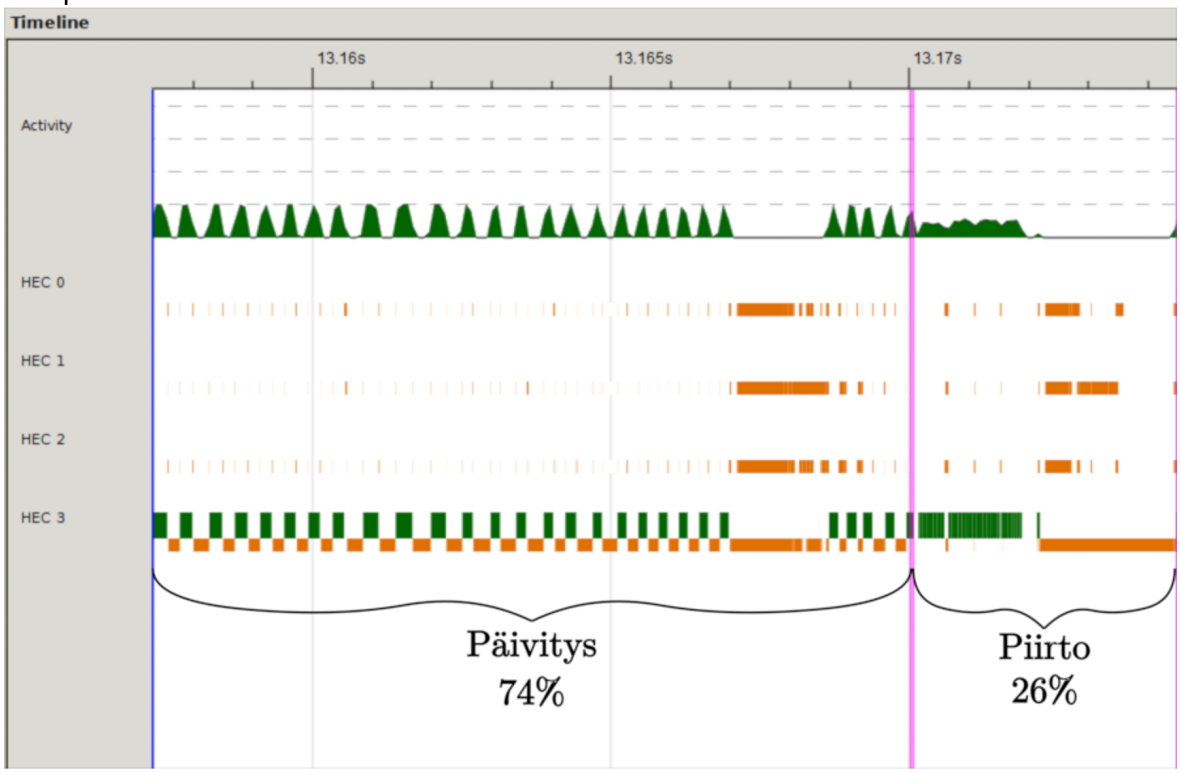
Kuviossa 12 on esitetty yksittäisen ruudunpäivityksen sisältö kummallakin funktiolla mitattuna. Kuviossa näkyy HEC-laskentakontekstit ja niiden suorittamat toiminnot tietyllä ajanhetkellä. Vihreät osuudet kuvaavat laskentaa ja oranssit roskienkeruuta. Kokonaisrasitus näykyy ylimpänä olevassa ”Activity”-graafissa.

`stepWires`-funktion tapauksessa kaikki laskenta tapahtui yhdellä laskentakontekstilla, koska sen laskenta ei rinnakkaistu. `parStepWires`-funktion tapauksessa jokaisesta objektin päivityksestä luotiin kipinä, jonka mikä tahansa vapaa HEC-laskentakonteksti voi ottaa rinnakkaisesti laskettavakseen. Tämä näkyy kuviossa päällekkäisinä vihreinä osioina. Rinnakkaistaminen kohdistettiin vain peliobjektien päivitykseen, jonka takia peliobjektien piirtäminen tapahtui kummassakin tapauksessa sarjassa.

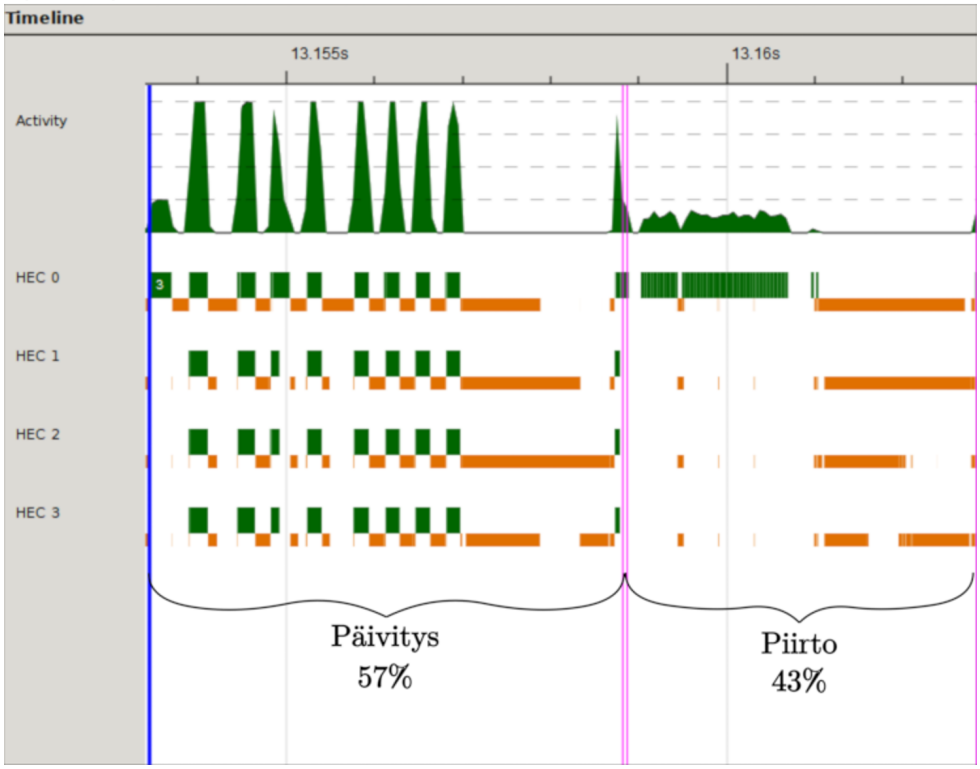
ThreadScopen avulla mitattiin, että `stepWires`-funktiolla peliobjektien päivitykseen meni 74% ja piirtämiseen 26% kokonaisajasta. `parStepWires`-funktion tapauksessa vastaavat luvut olivat 57% ja 43%. `parStepWires`-funktiota käytettäessä peliobjektien päivitykseen meni suhteellisesti vähemmän aikaa ja ruudunpäivitys tapahtui 1.83 kertaa nopeammin kuin `stepWires`-funktiolla.

Luvussa 2.4.5 esitetyn Amdahlin lain avulla voidaan laskea teoreettinen parannus suorituskykyyn suuremmilla ytimien määrillä. Kuviossa 13 on havainnollistettu Amhdahlin lakia soveltaen ThreadScopen avulla saatuja tuloksia. Rinnakkaistuva

stepWires

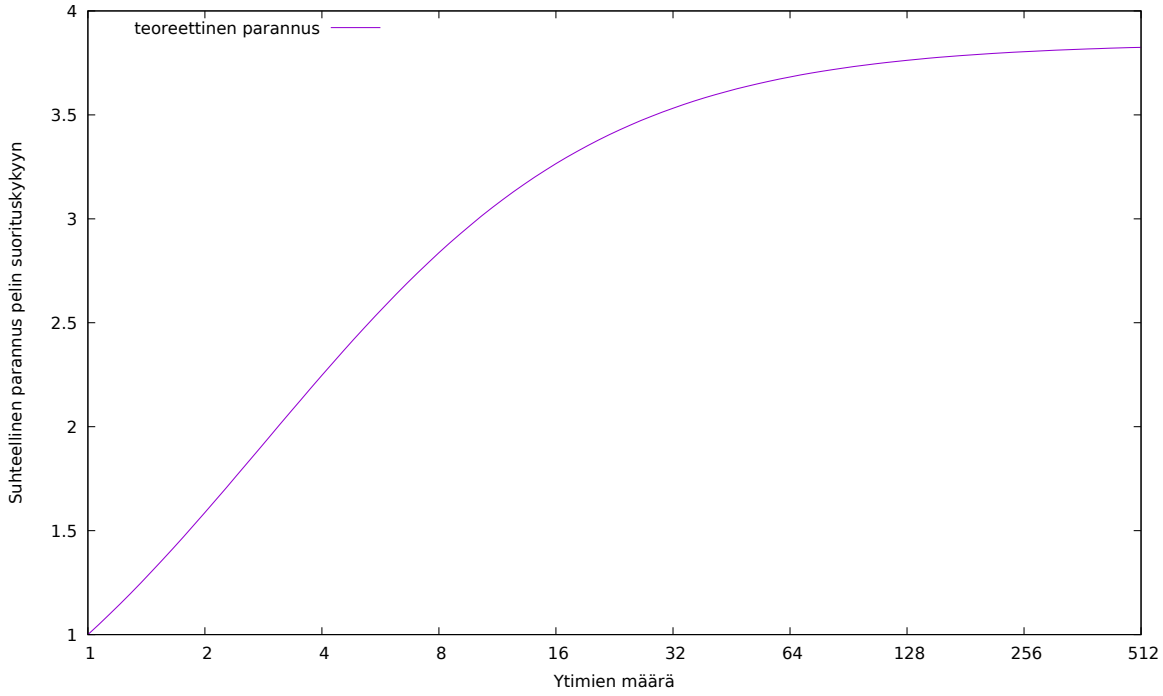


parStepWires



Kuvio 12. ThreadScopen näkymä `stepWires`- ja `parStepWires` -funktioille.

osuus on tässä tapauksessa peliobjektien päivittäminen. Amdahlin laissa olevan f-parametrin, eli rinnakkaistuvan osuuden arvoksi, saatiin ThreadScopen avulla 74%. Amdahlin laissa olevan s-parametrin suhteen oletettiin, että ytimien määrä on suoraan verrannollinen rinnakkaistetun osuuden suoritusnopeuden parantumiseen, jolloin esimerkiksi kahdella ytimellä rinnakkaistetun osuuden suoritusnopeus kaksinkertaistuu ja neljällä ytimellä nelinkertaistuu. Näiden oletusten takia kuviossa 13 näkyvä parannus on vain teoreettinen arvio, mutta siitä näkee silti tärkeän havainnon; vaikka ytimien määrää kasvattaa rajattomasti, suorituskyvyn parannus ei koskaan ole yli nelinkertainen.



Kuvio 13. Amdahlin lain mukainen teoreettinen parannus mittaustuloksista saaduilla parametreilla.

5 Pohdinta

Tutkimuksessa syntynyttä IT-artefaktia voidaan arvioida suunnittelutieteen suuntaviivojen mukaisesti. Tutkimuksessa tuotettiin tavoitteen mukainen merkityksellinen artefakti, joka on signaalifunktiokokoelman laskennan rinnakkaistamisen toteutus reaktiivisella funktio-ohjelmoinnilla tehtyjen pelien suorituskyvyn parantamiseen. Tuotetun artefaktin hyödyllisyyttä, laatua ja vaikutuksen arviointia tarkasteltiin suuntaviivojen mukaisesti määrittelemällä oikeanlaiset mittarit ja keräämällä ja analysoimalla artefaktista mitattua dataa. Artefaktia voitiin tässä tilanteessa arvioida sopivimmin suorituskyvyn näkökulmasta, jolloin suorituskyvyn parantumisesta kertovat mittaustulokset antoivat viitteitä artefaktin olevan toiminnoiltaan vaikuttava ja hyödyllinen.

Tieteellistä tarkkuutta ja täsmällisyyttä tässä tutkimuksessa tukivat huolellisesti suunniteltu ja toteutettu tutkimusprosessi, johon sisältyi artefaktin vaiheittainen rakentaminen ja mittausmenetelmien tarkka valitseminen. Lisäksi rinnakkaistamiseen liittyvä huolella valittu lähdeaineisto vaikutti syntyneen artefaktin laatuun ja oikeellisuuteen. Suunnitteluprosessi oli edennyt etsimisprosessin tavoin ongelmaa yksinkertaistamalla ja tarkastemalla sitä erilaisista näkökulmista päätyen lopulta toteutuksessa rinnakkaistamiseen.

Yleisellä tasolla voidaan todeta, että vaikka tutkimuksessa syntynyt artefakti täyttää IT-artefaktille ja suunnittelutieteelliselle lähdestymistavalle asetetut vaatimukset, on sille vielä tänä päivänä suhteellisen vähän käyttöä pelialalla. Funktionaalinen ohjelmointi soveltuu kuitenkin hyvin pelien ohjelmointiin ja funktionaalisen ohjelmoinnin yleistyessä voi tämän kaltaisille toteutuksille olla käyttöä tulevaisuudessa.

Tutkimuksessa suoritetun mittauksen perusteella tutkimuksessa toteutettu signaalifunktioiden laskentaa rinnakkaistava artefakti paransi testipelin suorituskykyä. Taulukon 2 (s. 51) perusteella neljällä säikeellä peliobjektien päivittäminen suoritettiin artefaktin avulla 1.8 kertaa nopeammin ja kahdeksalla 2.2 kertaa nopeammin. Toteutetun artefaktin heikkous on se, että sen tuoma parannus perustuu siihen, että

useiden peliobjektien päivittäminen on raskasta. Artefakti ei paranna suorituskykyä peleissä, joissa peliobjektien määrät ovat pieniä. Tuloksista havaittiin myös, että suorituskyky parantui vain marginaalisesti neljännen säikeen jälkeen. Saman suuntaisen havainnon on tehnyt Marlow (2013), jonka mukaan taustalla olevista muista prosesseista johtuen, optimaalisin määrä käytettäviä säikeitä olisi sama kuin prosessorin ytimien määrä.

Peliobjektien päivittämisen suorituskyvyn parantamista voi olla mahdollista viedä vielä pidemmälle. Rinnakkaistamisessa voi käyttää lohkottamistekniikkaa. Tämä parantaa suorituskykyä sopivilla säiemäärillä ja lohkon koolla, mutta lohkottaminen vaatii jonkin verran ylimääräistä työtä, joka aiheuttaa haittaa suorituskyvylle. Tutkimuksessa suoritetun mittauksen perusteella lohkottaminen paransi suorituskykyä pienillä säiemäärillä, mutta suuremmilla säiemäärillä suorituskyky joko heikkeni tai parani mielivaltaisesti. Lohkottamisen tehokkaasta käytöstä tekee haastavan se, että lohkon koko vaikuttaa suorituskykyyn ennalta arvaamattomalla tavalla ja sopivan lohkon koon joutuu usein määrittämään kokeilemalla.

Lohkottamisessa käytettiin valmista `Control.Parallel.Strategies`-kirjastoa, mutta lohkottamisen saattaa pystyä toteuttamaan tehokkaamminkin. Marlow (2013) on ehdottanut, että käyttämällä `Vector`-tietotyyppiä listan sijaan, voitaisiin mahdollisesti saavuttaa parempi suorituskyky, koska `Vector`-tietotyypin pilkkominen lohkoihin voidaan tehdä $\mathcal{O}(1)$ ajassa.

Tutkimusta suunniteltaessa mietittiin testipelin mittauksessa käytettyä laitteistoa. Mittauksessa päädyttiin käyttämään neljän ytimen prosessoria. Kyseisessä prosessorissa olevan *hyperthreading*-ominaisuuden ansiosta kahdeksaa säiettä pystyttiin laskemaan yhtäaikaisesti. Neliydinprosessorin valitsemiseen vaikutti myös se, että se vastaa paremmin tavallisen pelaajan laitteistoa kuin prosessori, jossa olisi huomattavasti enemmän ytimiä. Valitsemalla useamman kuin neljän ytimen prosessori, olisi voitu saavuttaa realistisemmat mittaustulokset suuremmille säiemäärille, mutta toisaalta Amhdahlin lain mukaan ytimien määrää rajattomasti kasvattamalla suorituskyky paranee vain tiettyyn rajaan asti.

Mittausten avulla saatiin selville, että Amdahlin lakia mukaillen, ytimiä rajattomasti kasvattamalla pelin suorituskyky ei voi teoriassa saavuttaa yli nelinkertaista parannusta (kuvio 13, s. 55). Testipelin tapauksessa artefaktilla pystyttiin rinnakkaistamaan vain peliobjektien päivittäminen.

Peleissä ruudunpäivitykseen sisältyy peliobjektien päivittämisen lisäksi niiden piirtäminen näytölle. Mikäli testipelin suorituskyvyn paranemista haluttaisiin kasvattaa, pitäisi koodista rinnakkaistaa myös peliobjektien piirtäminen. Tässä tutkimuksessa keskityttiin vain peliobjektien päivittämiseen, piirtämisen rinnakkaistamisen jäädessä vain pohdinnan tasolle.

Peliobjektien piirtämiseen käytetään usein ulkopuolisia C-kirjastoja, joiden käyttäminen Haskell-ohjelmoinnista vaatii ohjelmointikielten rajojen ylitystä foreign function interfacen (FFI) kautta, jonka rinnakkaistaminen ei ole yhtä triviaalia kuin peliobjektien päivittämisen rinnakkaistaminen. Rinnakkaistus on mahdollista kuitenkin, koska Haskell-ohjelmointikielessä voi tehdä FFI-kutsuja useista säikeistä yhtäaikaisesti (Marlow, Jones ja Thaller 2004).

6 Yhteenveto

Reaktiivinen ohjelmointi sopii hyvin interaktiivisten ohjelmien, kuten pelien kehittämiseen. Monissa peleissä suorituskyvyllä on suuri merkitys. Tutkimuksen tutkimusongelmana oli selvittää, miten suorituskykyä voidaan parantaa reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä. Tutkimuskysymyksinä oli reaktiivisella funktio-ohjelmoinnilla toteutettuihin peleihin liittyen seuraavat kysymykset: Millaisia tapoja on parantaa pelin suorituskykyä? Miten rinnakkaisuus toteutetaan ja miten se vaikuttaa suorituskykyyn?

Reaktiivisella funktio-ohjelmoinnilla tehdyissä peleissä suorituskyvyn parantamiseen on useita keinoja. Suorituskykyä voi parantaa esimerkiksi optimoimalla signaalifunktioista muodostuvaa verkkoa, vähentämällä laiskaa laskentaa tai rinnakkaislaskennalla. Reaktiivisessa funktio-ohjelmoinnissa peleissä olevia objekteja mallinnetaan signaalifunktioilla. Funktio-ohjelmoinnissa monet ohjelmat ovat luonnostaan helposti rinnakkaistettavissa ja Haskell-ohjelmointikielessä on hyvä tuki rinnakkaistamisen toteuttamiselle, joka teki siitä luontevan lähestymistavan suorituskyvyn parantamiselle. Tässä tutkimuksessa luotiin IT-artefakti, joka on lisättävissä Haskell-ohjelmointikielen Netwire-nimiseen reaktiiviseen funktio-ohjelmointikirjastoon yleiskäyttöiseksi funktioksi, jolla pystyy evaluoimaan signaalifunktiokokoelmia rinnakkaisesti.

Suorituskykymittausten perusteella tutkimuksessa toteutettu artefakti paransi testipelin suorituskykyä. Tutkimuksessa käytetyn artefaktin toimivuuden lisäksi tutkimus antaa viitteitä siihen, että suorituskyvyn parannusta pystyy vielä kasvattamaan esimerkiksi kehittämällä rinnakkaistamisen lohkottamistekniikkaa tai kiinnittämällä huomiota rinnakkaistettavan koodin osuuteen. Prosessorien ytimien määrät tulevat tulevaisuudessa kasvamaan. Tämä parantaa suorituskykyä vain tiettyyn rajaan asti ja paremman suorituskyvyn saavuttamiseksi suurempi osuus ohjelmasta tulisi rinnakkaistaa. Testauksessa käytetyssä pelissä voisi esimerkiksi saavuttaa paremman suorituskyvyn rinnakkaistamalla peliobjektien päivittämisen lisäksi myös niiden piirtämisen.

Jatkotutkimuksena voisi tutkia `Vector`-tietotyypin soveltamista signaalifunktioiden säilyttämiseen, `Vector`-kokoelman päivittämisen rinnakkaistamista ja lohkottamisen vaikutusta suorituskykyyn. Toinen jatkotutkimusaihe on reaktiivisella funktio-ohjelmoinnilla tehtyjen pelien piirtämisen rinnakkaistaminen.

Lähteet

Amdahl, Gene M. 1967. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". Teoksessa *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 483–485. Atlantic City, New Jersey: ACM. doi:10.1145/1465482.1465560.

———. 2013. "Computer architecture and Amdahl's law". *Computer* 46 (12): 38–46. ISSN: 0018-9162. doi:10.1109/MC.2013.418.

Armstrong, Joe. 2007. "A History of Erlang". Teoksessa *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, nide 12, 212. New York, USA: ACM. ISBN: 978-1-59593-766-7. doi:10.1145/1238844.1238850.

Bainomugisha, Engineer, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx ja Wolfgang De Meuter. 2013. "A Survey on Reactive Programming". *ACM Computing Surveys* 45 (4): 52:1–52:34. doi:10.1145/2501654.2501666.

Blackburn, Stephen M., Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss ym. 2008. "Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century". *Communications of the ACM* 51 (8): 83–89. ISSN: 0001-0782. doi:10.1145/1378704.1378723.

Boussinot, Frédéric. 1991. "Reactive C: An extension of C to program reactive systems". *Software: Practice and Experience* 21 (4): 401–428. doi:10.1002/spe.4380210406.

Cooper, Gregory. 2008. "Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language". Tohtorinväitöskirja, Brown University.

Courtney, Antony, Henrik Nilsson ja John Peterson. 2003. "The Yampa Arcade". Teoksessa *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, 7–18. ACM. doi:10.1145/871895.871897.

CPUID. 2017. *CPU-Z.* Viitattu 3. syyskuuta 2017. https://www.cpuid.com/softwares/cpu-z.html.

Elliott, Conal, ja Paul Hudak. 1997. “Functional reactive animation”. *SIGPLAN Not.* 32 (8): 263–273. ISSN: 0362-1340. doi:10.1145/258949.258973.

Ennals, Robert, ja Simon Peyton Jones. 2003. “Optimistic Evaluation : An Adaptive Evaluation Strategy for Non-Strict Programs”. *SIGPLAN Not.* 38:287–298. doi:10.1145/944746.944731.

GHC. 2014. *GHC/Memory Management*. Viitattu 24. lokakuuta 2017. https://wiki.haskell.org/GHC/Memory%7B%5C_%7DManagement.

———. 2015. *StrictPragma - GHC.* Viitattu 8. syyskuuta 2017. https://ghc.haskell.org/trac/ghc/wiki/StrictPragma.

GHC Team. 2015. *Running a compiled program - Glasgow Haskell Compiler.* Viitattu 1. syyskuuta 2017. https://downloads.haskell.org/%7B~%7Dghc/8.0.1/docs/html/users%7B%5C_%7Dguide/runtime%7B%5C_%7Dcontrol.html.

Gustafson, John L. 1988. “Reevaluating Amdahl’s Law”. *Communications of the ACM* 31 (5): 532–533. doi:10.1145/42411.42415.

Hackage. 2017a. *Control.Parallel.Strategies.* Viitattu 1. syyskuuta 2017. https://hackage.haskell.org/package/parallel-3.2.1.1/docs/Control-Parallel-Strategies.html.

———. 2017b. *Data.Monoid.* Viitattu 21. syyskuuta 2017. https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Monoid.html.

Harel, David, ja Amir Pnueli. 1985. *On the Development of Reactive Systems*, 477–498. Springer Berlin Heidelberg. ISBN: 978-3-642-82453-1. doi:10.1007/978-3-642-82453-1_17.

Hazelhurst, Scott. 2010. “Truth in advertising: Reporting performance of computer programs, algorithms and the impact of architecture”. *South African Computer Journal* 46:24–37. ISSN: 2313-7835. doi:10.18489/sacj.v46i0.50. http://sacj.cs.uct.ac.za/index.php/sacj/article/view/50.

Herlihy, Maurice, ja Victor Luchangco. 2008. “Distributed computing and the multicore revolution”. *ACM SIGACT News* 39 (1): 62. ISSN: 01635700. doi:10.1145/1360443.1360458.

Hevner, Alan R., Salvatore T. March ja Jinsoo Park. 2004. “Design Science in Information Systems Research”. *MIS Q.* 28 (1): 75–105. http://dl.acm.org/citation.cfm?id=2017212.2017217.

Hill, Mark D., ja Michael R. Marty. 2017. “Retrospective on amdahl’s law in the multicore era”. *Computer* 50 (6): 12–14. ISSN: 0018-9162. doi:10.1109/MC.2017.164.

Hirsijärvi, Sirkka, Pirkko Remes ja Paula Sajavaara. 2001. *Tutki ja kirjoita.* 7. painos. Vantaa: Tummavuoren kirjapaino Oy. ISBN: 951-26-4618-8.

Hudak, Paul, ja Joseph Fasel. 1992. “A Gentle Introduction to Haskell”. *SIGPLAN Not.* 27 (5): 1–52. doi:10.1145/130697.130698. https://www.haskell.org/tutorial/.

Hudak, Paul, T Johnsson, M Jones ja J Launchbury. 2003. *Haskell 98 Language and Libraries: The Revised Report.* Toimittanut Simon Peyton Jones. Cambridge University Press. ISBN: 9780521826143. doi:10.2277/0521826144.

Hughes, John. 2004. “Programming with Arrows”. Teoksessa *Advanced Functional Programming*, toimittanut Varmo Vene ja Tarmo Uustalu, 73–129. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-28540-3. doi:10.1007/11546382.

Intel. 2017. *Intel® Core™ i7-3770K Processor (8M Cache, up to 3.90 GHz) Product Specifications.* Viitattu 26. elokuuta 2017. http://ark.intel.com/products/65523/Intel-Core-i7-3770K-Processor-8M-Cache-up-to-3%7B%5C_%7D90-GHz.

Jones, Jr. Don, Simon Marlow ja Satnam Singh. 2009. “Parallel Performance Tuning for Haskell”. Teoksessa *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, 81–92. Edinburgh, Scotland: ACM. ISBN: 978-1-60558-508-6. doi:10.1145/1596638.1596649.

Kalibera, Tomas, ja Richard Jones. 2013. “Rigorous Benchmarking in Reasonable Time”. *SIGPLAN Not.* 48 (11): 63–74. ISSN: 0362-1340. doi:10.1145/2555670.2464160.

Kyyppö, Jorma, ja Mika Vesterholm. 2015. *Java-ohjelmointi.* 9. painos. Helsinki: Talentum Media Oy. ISBN: 978-952-14-2520-2.

Liu, Hai, Eric Cheng ja Paul Hudak. 2009. “Causal commutative arrows and their optimization”. *SIGPLAN Not.* 44 (9): 35–46. doi:10.1145/1631687.1596559.

Marlow, Simon. 2013. *Parallel and Concurrent Programming in Haskell.* O’Reilly Media. ISBN: 978-1-4493-3594-6. http://chimera.labs.oreilly.com/books/1230000000929.

Marlow, Simon, Tim Harris, Roshan P. James ja Simon Peyton Jones. 2008. “Parallel Generational-copying Garbage Collection with a Block-structured Heap”. *Proceedings of the 7th International Symposium on Memory Management:* 11–20. doi:10.1145/1375634.1375637. http://portal.acm.org/citation.cfm?doid=1375634.1375637.

Marlow, Simon, Simon Peyton Jones ja Wolfgang Thaller. 2004. “Extending the Haskell Foreign Function Interface with Concurrency”. *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell:* 22–32. doi:10.1145/1017472.1017479. http://dl.acm.org/citation.cfm?id=1017472.1017479.

Marlow, Simon, Simon Peyton Jones ja Satnam Singh. 2009. “Runtime Support for Multicore Haskell”. *SIGPLAN Not.* 44 (9): 65–78. ISSN: 0362-1340. doi:10.1145/1631687.1596563. http://portal.acm.org/citation.cfm?doid=1631687.1596563.

Maruseac, Mihai. 2017. *Haskell Communities and Activities Report.* Viitattu 24. lokakuuta 2017. https://www.haskell.org/communities/05-2017/report.pdf.

Microsoft. 2017a. *Acquiring high-resolution time stamps (Windows).* Viitattu 24. lokakuuta 2017. https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx.

Microsoft. 2017b. *FILETIME structure (Windows).* Viitattu 24. lokakuuta 2017. https://msdn.microsoft.com/fi-fi/library/windows/desktop/ms724284(v=vs.85).aspx.

Mytkowicz, Todd, Amer Diwan, Matthias Hauswirth ja Peter F. Sweeney. 2009. "Producing Wrong Data Without Doing Anything Obviously Wrong!" *SIGPLAN Not.* 44 (3): 265–276. ISSN: 0362-1340. doi:10.1145/1508284.1508275.

Neogames. 2016. *Finnish Game Industry Report.* Viitattu 25. lokakuuta 2017. http://www.neogames.fi/wp-content/uploads/2017/04/Finnish-Game-Industry-Report-2016%7B%5C_%7Dweb%7B%5C_%7D070529.pdf.

Paterson, Ross. 2001. "A New Notation for Arrows". *SIGPLAN Not.* 36 (10): 229–240. ISSN: 0362-1340. doi:10.1145/507546.507664.

Perez, Ivan. 2017. *Yampa/Switches.hs.* Viitattu 20. lokakuuta 2017. https://github.com/ivanperez-keera/Yampa/blob/fb8edeef34470d509620bdab1daee6dac4107148/src/FRP/Yampa/Switches.hs.

Perez, Ivan, Manuel Bärenz ja Henrik Nilsson. 2016. "Functional Reactive Programming, Refactored". Teoksessa *Proceedings of the 9th International Symposium on Haskell*, 33–44. Nara, Japan: ACM. doi:10.1145/2976002.2976010.

Peterson, John, Valery Trifonov ja Andrei Serjantov. 2000. *Parallel Functional Reactive Programming*, 16–31. Springer Berlin Heidelberg. ISBN: 978-3-540-46584-3. doi:10.1007/3-540-46584-7_2.

Piro, Christopher, ja Eugene Letuchy. 2009. "Functional programming at Facebook". Teoksessa *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming Functional Programming As a Means, Not an End*, 1. New York, New York, USA: ACM Press. ISBN: 9781605589435. doi:10.1145/1668113.1668120. http://dl.acm.org/citation.cfm?doid=1668113.1668120.

Pucella, Riccardo R. 1998. "Reactive programming in Standard ML". *Proceedings of the 1998 International Conference on Computer Languages:* 48–57. ISSN: 1074-8970. doi:10.1109/ICCL.1998.674156.

Sculthorpe, Neil, ja Henrik Nilsson. 2010. “Keeping calm in the face of change”. *Higher-Order and Symbolic Computation* 23 (2): 227–271. ISSN: 1573-0557. doi:10.1007/s10990-011-9068-x.

Sert, Cetin. 2016a. *clock/hs_clock_win32.c.* Viitattu 24. lokakuuta 2017. https://github.com/corsis/clock/blob/420943107101d8808af5d57a10a92402f34b39e2/cbits/hs%7B%5C_%7Dclock%7B%5C_%7Dwin32.c.

———. 2016b. *System.Clock.* Viitattu 24. lokakuuta 2017. https://hackage.haskell.org/package/clock-0.7.2/docs/System-Clock.html.

Sutter, Herb, ja James Larus. 2005. “Software and the concurrency revolution”. *Queue* 3 (7): 54. ISSN: 15427730. doi:10.1145/1095408.1095421. http://portal.acm.org/citation.cfm?doid=1095408.1095421.

Söylemez, Ertugrul. 2017. *netwire: Functional reactive programming library.* Viitattu 24. huhtikuuta. https://hackage.haskell.org/package/netwire.

Ungar, David. 1984. “Generation Scavenging: A Non-disruptlve High Performnance Storage Reclamation Algorithm”. *SIGSOFT Software Engineering Notes* 9 (3): 157–167. ISSN: 0163-5948. doi:10.1145/390010.808261.

Wadler, Philip. 1997. “How to Declare an Imperative”. *ACM Computing Surveys* 29 (3): 240–263. ISSN: 0360-0300. doi:10.1145/262009.262011.