Simo Rinne

Funktio-ohjelmoinnin hyödyntäminen peliohjelmoinnissa

Tietotekniikan kandidaatintutkielma

28. huhtikuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Simo Rinne

Yhteystiedot: simo.e.rinne@student.jyu.fi

Ohjaaja: Anneli Heimbürger

Työn nimi: Funktio-ohjelmoinnin hyödyntäminen peliohjelmoinnissa

Title in English: Utilization of functional programming in game programming

Työ: Kandidaatintutkielma

Sivumäärä: 22+0

Tiivistelmä: Pelialalla käytetään suurimmaksi osaksi imperatiivisia ohjelmointikieliä. Tämän kandidaatintutkielman tavoitteena on tutkia mitä hyötyjä ja haittoja funktio-ohjelmoinnin käyttö tuo peliohjelmointiin. Funktio-ohjelmoinnin käyttö tekee yhtäaikaisen laskennan ja rinnakkaislaskennan käytöstä helpompaa. Puhtaiden funktioiden käytöllä voidaan vähentää ohjelmassa olevia bugeja ja ohjelman toiminnan päättely on helpompaa. Reaktiivinen funktio-ohjelmointi vaikuttaa lupaavalta tavalta tehdä pelejä.

Avainsanat: funktionaalinen ohjelmointi, peliohjelmointi, reaktiivinen funktio-ohjelmointi

Abstract: Imperative programming languages are most commonly used in the game industry. The objective of this bachelor's thesis is to investigate what advantages and disadvantages functional programming brings to game programming. It is easier to write concurrent and parallel programs using functional programming languages. Usage of pure functions can reduce bugs and makes the program easier to reason about. Functional reactive programming seems to be a promising way to program games.

Keywords: functional programming, game programming, functional reactive programming

Kuviot

Sisältö

1 Johdanto

Tietokoneella ja konsoleilla pelattavat videopelit ovat yksi voimakkaimmista ja vaikutusvaltaisimmista taiteen muodoista. Videopeleistä on tullut nopeiten kasvava viihdeala, jonka liikevaihto kilpailee elokuva- ja musiikkialan liikevaihdon kanssa. (Gershenfeld, Loparco & Barajas 2003.)

Videopelit koostuvat kolmesta osa-alueesta: peliohjelmoinnista, pelin suunnittelusta ja pelin sisällöstä. Sisältöön kuuluu grafiikka, 3D-mallit, tarina, ääniefektit ja musiikki. Suunnitteluun kuuluu pelimekaniikan, kenttien, visuaalisen tyylin ja käyttöliittymän suunnittelu. Peliohjelmointiin kuuluu simulointi, tekoäly, fysiikkamallinnus sekä pelin renderöinnin ja äänentoiston ohjelmointi. Jokainen näistä kolmesta osa-alueesta on itsenäinen komponentti ja yhdessä ne muodostavat interaktiivisen kokemuksen. (Ted Tschang 2003.)

Funktio-ohjelmointi on ollut suosittua tietojenkäsittelytieteessä jo vuosikymmeniä, mutta se on otettu käyttöön oikeisiin ohjelmiin vasta viime aikoina (Hinsen 2009). Hinsen (2009) on pohtinut, että funktio-ohjelmoinnin hitaalle yleistymiselle on monta syytä, mutta kaksi syytä on erityisen tärkeitä. Ensimmäinen niistä on se, että funktionaalinen ohjelmointi poikkeaa hyvin paljon imperatiivisesta ohjelmointityylistä, ja tästä johtuen funktio-ohjelmoinnin oppiminen koetaan haastavana. Toinen syy on se, että tietokoneiden laitteistot tukevat paremmin imperatiivista ohjelmointimallia, ja tästä johtuen imperatiivisilla ohjelmointikielillä tehtyjä ohjelmia on helpompi kääntää tehokkaiksi konekielisiksi ohjelmiksi.

McGill (2008) on tehnyt kyselytutkimuksen, jonka mukaan pelialalla eniten käytetty kieli on C++. Muita paljon käytettyjä kieliä olivat muunmuassa C#, Java ja Lua. Kaikki näistä ovat imperatiivisia ohjelmointikieliä ja tämän tutkimuksen tarkoituksena on tutkia, kuinka hyvin funktionaalinen ohjelmointiparadigma soveltuu peliohjelmointiin.

Luvussa 2 tutustutaan funktionaaliseen peliohjelmointiin käyttäen Haskell-ohjelmointikieltä sekä peleihin liittyviä esimerkkejä. Luvussa 3 pohditaan funktio-ohjel-

moinnin hyötyjä ja haittoja peliohjelmoinnissa. Luvussa 4 tutustutaan reaktiiviseen funktio-ohjelmointiin ja sen soveltamiseen peliohjelmoinnissa. Lukuun 5 on koottu johtopäätöksiä, omia pohdintoja ja aiheita jatkotutkimukseen.

Tässä työssä termi *functional reactive programming* on käännetty suomenkielelle muotoon *reaktiivinen funktio-ohjelmointi*, koska *funktionaalinen reaktiivinen ohjelmointi* kuulostaa omituiselta.

2 Johdatus funktionaaliseen peliohjelmointiin

Monilla ohjelmointikielillä pystyy ohjelmoimaan funktionaalisella tyylillä soveltaen funktio-ohjelmoinnin periaatteita. Tässä luvussa esitellään lyhyesti funktio-ohjelmointia käyttäen Haskell-ohjelmointikieltä ja peleihin liittyviä esimerkkejä. Lähdekoodi listauksissa käytetään hieman matemaattisempaa notaatiota, koska se on vakiintunut tapa esittää funktionaalista koodia akateemisissa teksteissä.

2.1 Funktiot

Haskelissa ja muissakin funktionaalisissa ohjelmointikielissä funktiot ovat ohjelmien tärkeimpiä rakennuspalikoita. Haskelissa funktiot kirjoitetaan normaalisti curry muotoon. Curryamisella tarkoitetaan sitä, että funktio, joka ottaa kaksi parametria, voidaan esittää funktiona, joka ottaa yhden parametrin ja palauttaa funktion, joka ottaa yhden parametrin. Haskelissa on myös anonyymejä funktioita, eli lambdoja. Kaikki funktiot on mahdollista kirjoittaa infiksi tai prefiksi muodossa. (Hudak, Hughes, Peyton Jones & Wadler 2007.)

Listauksessa 2.1 on esitetty esimerkit kahden funktion määrityksistä. Aluksi kirjoitetaan funktion tyyppimääritelmä. Tyyppimääritelmässä nuolen vasemmalla puolella on lähtöjoukko ja oikealla puolella on maalijoukko. Tyyppimääritelmässä olevat nuolet ovat oikealle assosiatiivisia. Tyyppimääritelmän seuraaville riveille tulee funktion toteutus. Funktion toteutukseen kirjoitetaan ensin funktion nimi, jonka jälkeen tulee parametrit välilyönneillä erotettuna. Sen jälkeen tulee yhtäsuuri kuin -merkki ja lopuksi funktion sisältö. Listauksessa 2.1 oleva funktio `f` ottaa yhden kokonaisluku parametrin ja palauttaa kokonaisluvun. Funktiolle `g` voi antaa kaksi parametria ja lopputuloksena on kokonaisluku.

Listaus 2.1: Funktioiden määrittely

```
f :: Int → Int
f x = 1 + x

g :: Int → Int → Int
g x y = x + y
```

Haskell-ohjelmointikielessä kaikki funktiot ottavat oikeasti vain yhden parametrin. Funktioiden tyyppimääritelmässä olevat nuolet ovat oikealle assosiatiivisia, joten listauksessa 2.1 olevan funktion `g` tyypin voi ajatella olevan `Int → (Int → Int)`, eli funktio joka ottaa kokonaisluvun ja palauttaa funktion. Esimerkiksi jos `g`-funktiolle antaa yhden parametrin[1], niin lopputuloksena on funktio, jonka tyyppi on `Int → Int`, eli funktio, joka ottaa yhden kokonaisluvun ja palauttaa kokonaisluvun. Haskell-ohjelmointikielessä kahden parametrin anto tarkoittaa siis sitä, että ensin annetaan yksi parametri, ja lopputuloksena tulevalle funktiolle annetaan se toinen parametri.

Funktiokutsuja ilmaistaan asettamalla funktio ja parametrit vierekkäin välilyönneillä erotettuna (Tibell 2010). Toisin kuin imperatiivisissa ohjelmointikielissä, funktiokutsuun ei laiteta sulkuja. Esimerkiksi funktion `f` kutsumista kahdella parametrilla `x` ja `y` kuvataan merkinnällä `f x y`. Funktiokutsut ovat vasemmalle assosiatiivisia, eli esimerkiksi lauseke `f x y` on identtinen lausekeen `(f x) y` kanssa (Hudak ym. 2007). Tällä kirjoitustyylillä saadaan tiivistä ja ilmaisuvoimaista koodia (Hudak ym. 2007).

2.2 Tyypit

Tietotyypit ovat funktioiden lisäksi tärkeässä asemassa funktionaalisissa ohjelmointikielissä. Listauksessa 2.2 on esitetty esimerkki ristinolla-pelin tietotyypeistä. `Merkki` voi olla joko `Risti`, `Nolla` tai `Tyhjä`. `PeliTila`:ssa oleva `[Merkki]` tarkoittaa listaa, jossa on `Merkki` tyyppisiä alkioita. Esimerkki meneillään olevasta pelistä voisi olla vaik-

[1] Jos tämä annettu parametri olisi esimerkiksi luku 1, niin lopputulosena oleva funktio olisi identtinen listauksessa 2.1 olevan funktion `f` kanssa.

kapa `Peli [Risti, Nolla, Tyhjä, Tyhjä, Tyhjä, Risti, Nolla, Nolla, Tyhjä]`, joka kuvaisi kuvion 1 pelitilaa.

Listaus 2.2: Tietotyypit ristinolla-peliin

```
data Merkki = Risti | Nolla | Tyhjä
data PeliTila = Peli [Merkki] | PeliOhi
```

Haskell-ohjelmointikielessä lista on rekursiivinen tietorakenne. Lista on määritelty siten, että se on joko tyhjä lista, jota kuvaa merkintä `[]`, tai sitten siinä on alkio `x`, jota seuraa lista `xs`, jota kuvataan merkinnällä `x:xs`. (Tibell 2010.) Esimerkiksi listaa, jossa on vain alkio `x`, kuvaa merkintä `x:[]`. Listaa, jossa on kolme alkiota `x`, `y` ja `z` kuvaa merkintä `x:(y:(z:[]))`. Listojen määrittelyä on helpotettu syntaksin suhteen siten, että vastaavan listan voi esittää myös muodossa `[x, y, z]`.

Yhdelle funktiolle voi kirjoittaa monta määritelmää käyttämällä mallin sovitusta (engl. *pattern matching*) (Hudak ym. 2007, Tibell 2010). Funktion parametriä yritetään sovittaa eri määritelmiin ylimmästä määritelmästä alkaen, kunnes sopiva määritelmä löytyy. Ohjelman suoritus keskeytyy jos sopivaa määritelmää ei löydy.

Listaus 2.3: Esimerkki mallin sovituksesta (Tibell 2010)

```
sum []     = 0
sum (x:xs) = x + sum xs
```

Esimerkiksi listauksessa 2.3 olevalle `sum`-funktiolle on annettu kaksi määritelmää mallin sovitusta hyödyntäen. Funktion parametria sovitetaan ensin ylempään määritelmään, johon sopii vain tyhjä lista. Jos parametri ei olekaan tyhjä lista, niin sitä sovitetaan alempaan määritelmään siten, että `x`:n arvoksi tulee listan ensimmäinen

X	O	
		X
O	O	

Kuvio 1: Ristinolla-pelin tilanne

alkio ja loput listasta menee `xs`:ään.

2.3 Tyyppiluokat

Tyyppiluokat kuvaavat operaatioiden joukkoa, jotka jokainen tyyppiluokkaan kuuluva tyyppi toteuttaa (Hudak ym. 2007). Haskell-ohjelmointikielessä on Show-tyyppiluokka, johon kuuluvien tyyppien pitää toteuttaa show-funktio, joka muuttaa tyypin merkkijonoksi. Show-tyyppiluokan määritelmä on esitetty listauksessa 2.4. `Merkki`-tyypin voi sisällyttää Show-tyyppiluokkaan tekemällä sille instanssin, jossa on toteutettu kaikki Show-tyyppiluokassa määritetyt funktiot. Esimerkki tästä on esitetty listauksessa 2.5.

Listaus 2.4: Show-tyyppiluokka

```
class Show a where
    show :: a → String
```

Listaus 2.5: Show-tyyppiluokan toteutus Merkki-tietotyypille

```
instance Show Merkki where
    show Risti = "X"
    show Nolla = "O"
    show Tyhjä = " "
```

Haskell ohjelmointikielessä tyyppiluokkia voi tehdä konkreettisien tyyppien lisäksi myös keskeneräisille tyyppikonstruktoreille, joille ei ole annettu kaikkia parametreja (Hudak ym. 2007). Tyyppikonstruktoreita on esimerkiksi listauksessa 2.7 olevan `Maybe`-tyypin määrittelyrivillä olevan yhtäsuuri kuin -merkin vasemmalla puolella oleva `Maybe`. Yhtäsuuri kuin -merkin oikealla puolella on datakonstruktorit `Nothing` ja `Just`.

Haskell-ohjelmointikielessä on myös mahdollisuus tehdä tyyppiluokkia tyypeille, joiden tyyppikonstruktorissa on useita parametreja. Tämä ominaisuus ei kuulu alkuperäiseen Haskell 98 -standardiin, mutta Glasgow Haskell Compiler tukee sitä

kielilaajennuksien kautta. (Hudak ym. 2007.)

2.4 Monadit

Monadeilla voi tuoda hallitusti sivuvaikutuksia puhtaasti funktionaalisiin ohjelmointikieliin (Wadler 1997). Sivuvaikutuksiin perehdytään tarkemmin luvussa 3.1. Wadler (1997) on kirjoittanut, että MacLanen (1971) mukaan monadi on käsite, joka on lähtöisin kategoriateoriasta. Wadler (1997) on kirjoittanut myös, että Eugenio Moggin (1989, 1991) mukaan monadeilla voi mallintaa monia ohjelmointikielten ominaisuuksia, kuten ohjelman tilaa, poikkeuksia, jatkumoa (engl. *continuation*), sekä interaktiota.

Haskell-ohjelmointikielessä on `Monad`-tyyppiluokka, joka on esitetty listauksessa 2.6. Tyyppien tyyppiä kuvataan tässä tekstissä termillä laji (engl. *kind*), jotta voidaan erotella arvotason tyypit ja tyyppitason tyypit. Listauksessa 2.6 esiintyy tyyppitason muuttuja `m`, jonka laji on $* \rightarrow *$. Tämä laji vastaa tyyppitasolla olevaa tyyppifunktiota. Lajimerkinnöissä $*$ vastaa jotain yksittäistä mielivaltaista tyyppiä, joten $*$ lausutaan siis tyyppinä. Esimerkiksi merkintä $* \rightarrow *$ lausutaan "tyypiltä tyypille". Listauksessa 2.7 esiintyvä `Maybe`-tyyppi on lajiltaan $* \rightarrow *$, ja tästä johtuen `Maybe`-tyyppistä voi tehdä `Monad`-tyyppiluokan instanssin (Hudak ym. 2007). `Maybe`-tyypin `Monad`-instanssin toteutus on esitetty myös listauksessa 2.7.

Listaus 2.6: Monad-tyyppiluokka (Hudak ym. 2007)

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

Listaus 2.7: Maybe-tyyppi ja sen Monad instanssi (Hudak ym. 2007)

```
data Maybe a =  Nothing | Just a

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

2.5 Pelitilan päivitys funktionaalisesti

Puhtaasti funktionaalisissa ohjelmointikielissä ei ole muuttujia, vaan kaikki arvot ovat vakioita (Hinsen 2009). Tästä johtuen pelitilan päivitys ilman sivuvaikutuksia pitää tehdä siten, että funktiolle annetaan pelin tila ja funktio palauttaa uuden päivitetyn tilan. Listauksessa 2.8 on esitetty esimerkki funktiosta, jolla voi päivittää listauksessa 2.2 määritettyä `PeliTila`-tietotyyppiä. Funktio ei ota kantaa siihen, että onko uuden pelimerkin alla jo toinen pelimerkki. Haskelissa ei ole silmukoita, joten `laitaMerkki`-funktion sisällä määritelty `laita`-funktio on toteutettu rekursiivisesti.

Muuttujien ja silmukoiden puuttuminen tulee usein suurena yllätyksenä niille, jotka ovat ohjelmoineet imperatiivisilla ohjelmointikielillä ja opettelevat funktio-ohjelmointia (Hinsen 2009). Jos pelin tila on suuri tietorakenne, niin uuden pelitilan jatkuva uudelleen luominen voi alkaa arveluttamaan suorityskyvyn ja muistinkäytön kannalta. Nokkela kääntäjä pystyy kuitenkin optimoimaan koodin siten, että pelin tilaa ei luoda kokonaan uudestaan, vaan viitataan vanhassa pelitilassa olevaan tietoon.

Listaus 2.8: Pelitilan päivitys funktionaalisesti

```
laitaMerkki :: PeliTila → (Int, Int) → Merkki → PeliTila
laitaMerkki (Peli merkit) (x, y) uusiMerkki = Peli (laita merkit 0)
    where laita (merkki:loput) i
            | i == y * 3 + x = uusiMerkki : loput                    (1)
            | otherwise      = merkki     : laita loput (i + 1)      (2)
```

Havainnoillistetaan listauksessa 2.8 olevan funktion toimintaa päivittämällä kuviota 1 vastaavaa pelitilaa siten, että yläoikealla olevaan ruutuun, jota vastaa koordinaatti (2, 0), laitetaan risti. Listauksessa 2.9 on laskettu vaiheittain mitä laitaMerkki-funktiokutsun lopputulokseksi tulee, kun sille annetaan vanha pelitila, uuden merkin sijainti, sekä uusi merkki. Tilansäästön vuoksi koko pelitilaa ei näytetä kokonaisuudessaan. Poisjätetyt osat on merkitty kolmella peräkkäisellä pisteellä. Listauksen 2.9 viimeisellä rivillä olevaa pelitilaa vastaa kuvio 2.

Listaus 2.9: Pelitilan päivityksen havainnoillistaminen

```
 laitaMerkki (Peli [Risti, Nolla, Tyhjä, Tyhjä, ...]) (2, 0) Risti
⇔ (korvataan laitaMerkki omalla määritelmällään)
 Peli (laita [Risti, Nolla, Tyhjä, Tyhjä, ...] 0)
⇔ (käytetään laita-funktion määritelmää 2, koska 0 ≠ 0 * 3 + 2)
 Peli (Risti : laita [Nolla, Tyhjä, Tyhjä, ...] 1)
⇔ (käytetään laita-funktion määritelmää 2, koska 1 ≠ 0 * 3 + 2)
 Peli (Risti : Nolla : laita [Tyhjä, Tyhjä, ...] 2)
⇔ (käytetään laita-funktion määritelmää 1, koska 2 = 0 * 3 + 2)
 Peli (Risti : Nolla : Risti : [Tyhjä, ...])
⇔ (muutetaan lista syntaktisesti mukavampaan muotoon)
 Peli [Risti, Nolla, Risti, Tyhjä, ...]
```

X	O	X
		X
O	O	

Kuvio 2: Päivitetty ristinolla-pelin tilanne

3 Funktio-ohjelmoinnin hyödyt ja haitat peliohjelmoinnissa

Tässä luvussa esitellään yleisellä tasolla funktio-ohjelmoinnin hyötyjä ja haittoja peliohjelmoinnissa. Funktio-ohjelmointi tarjoaa peliohjelmointiin erilaisia hyötyjä, kuten sivuvaikutuksien hallintaa ja rinnakkaislaskentaa, mutta peliohjelmointi ei kuitenkaan ole täysin ongelmatonta funktionaalisilla ohjelmointikielillä. Olen huomannut oman kokemuksen pohjalta, että ongelmia tuottavat ainakin laiskasta laskennasta johtuvat tilavuodot.

3.1 Sivuvaikutuksien hallinta ja tiedon läpinäkyvyys

Puhtaat funktiot palauttavat aina saman lopputuloksen samalla parametrilla ja ne eivät voi sisältää sivuvaikutuksia. Sivuvaikutuksia ovat esimerkiksi ohjelman tilan muuttaminen ja käyttäjän kanssa kommunikointi. Sivuvaikutukset ovat pelien kannalta tärkeitä, koska pelit ovat sovelluksia, jotka ovat jatkuvassa interaktiossa pelaajan kanssa. Haskell-ohjelmointikielessä sivuvaikutuksia voi mallintaa monadeilla, joita on kuvattu tarkemmin luvussa 2.4.

Puhtaita funktioita yhdistelemällä voi muodostaa isomman lausekkeen, joka on edelleen sivuvaikutuksista vapaa. Tälläistä lauseketta sanotaan tiedollisesti läpinäkyväksi (engl. *referentially transparent*), koska sen voi vapaasti korvata lopputuloksellaan vaikuttamatta ohjelman käytökseen. Puhtaat funktiot ovat virheettömämpiä, koska niiden suoritusjärjestyksellä ei ole merkitystä ja tiedon läpinäkyvyys auttaa päättelemään ohjelman toimintaa. (Wadler 1997.)

Suurin syy ohjelmien epäonnistumiselle on huono laadunvarmistus. Bugien etsiminen ja niiden korjaaminen on kallein ja eniten aikaa vievin osa-alue ohjelmistokehityksessä (Jones 1995). Funktionaalisten ohjelmointikielten käyttö voisi olla yksi ratkaisu siihen. Hughes (1989) on sitä mieltä, että suuren osan ohjelmissa esiintyvistä bugeista voi estää käyttämällä funktioita, joilla ei ole hallitsemattomia sivuvaiku-

tuksia. Peleissä on yleensä jonkinlainen tila, joka muuttuu jatkuvasti pelin edetessä. Isommissa peleissä pelin tila voi olla suurikin tietorakenne. Jos epäpuhtaat funktiot voivat vapaasti aiheuttaa muutoksia siihen, niin ohjelman toiminnan seuraaminen sekä debuggaus saattaa olla haastavaa, erityisesti ohjelmissa, jotka käyttävät useita säikeitä.

3.2 Yhtäaikainen laskenta ja rinnakkaislaskenta

Rinnakkaislaskentaa ja yhtäaikaista laskentaa hyödyntäviä ohjelmia on helpompi tehdä funktionaalisilla ohjelmointikielillä kuin imperatiivisilla ohjelmointikielillä (Hinsen 2009). Rinnakkaislaskennalla (engl. *parallelism*) tarkoitetaan yhden isomman laskutoimituksen jakamista pienemmiksi osiksi, joita voidaan laskea prosessorin eri ytimillä samaan aikaan. Yhtäaikaisella laskennalla (engl. *concurrency*) tarkoitetaan ohjelman suorituksen jakamista säikeiksi. (Hinsen 2009.) Glasgow Haskell Compiler (GHC) tukee transaktiomuistia, joka helpottaa säikeiden välistä kommunikaatiota ja parantaa ohjelman modulaarisuutta (Harris, Marlow, Peyton Jones & Herlihy 2008).

Pelien monimutkaistuessa laskentatehon tarve kasvaa myöskin. Tämän tarpeen pystyy täyttämään helpommin funktionaalisilla ohjelmoinikielillä kuin imperatiivisilla ohjelmointikielillä. Rinnakkaislaskenta ei kuitenkaan ole mikään hopealuoti laskentatehon saamiseen, koska ohjelman nopeutusta rajoittaa aika, joka tarvitaan ohjelman peräkkäisten osuuksien suorittamiseen (Tibell 2010). Tämä rajoitus on nimeltään Amdahlin laki (Tibell 2010).

3.3 Tilavuodot

Tilavuoto tapahtuu, kun aletaan laskemaan laskutoimitusta liian laiskasti. Laiskalla laskennalla tarkoitetaan sitä, että lausekkeen lopullinen arvo lasketaan vasta kun sitä tarvitaan (Tibell 2010). Jos lauseketta ei tarvita ollenkaan, niin sen lopullista arvoa ei lasketa koskaan. Esimerkki tilavuodon aiheuttavasta funktiosta on esitetty Haskell-ohjelmointikielellä listauksessa 3.1.

Listaus 3.1: Esimerkki tilavuodon aiheuttavasta funktiosta (Yang 2011)

```
f []     c = c
f (x:xs) c = f xs (c + 1)
```

Naiivi kääntäjä ei huomaa, että lauseketta `(c + 1)` ei kannata laskea laiskasti, eli jättää laskematta, kunnes on ihan pakko. Ongelma ilmenee selkeämmin listauksessa 3.2, kun lasketaan vaihe kerrallaan, mitä esimerkiksi lausekkeen `f [1..10] 0` arvoksi tulee.

Listaus 3.2: Tilavuodon havainnoillistaminen

```
f [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 0
f [2, 3, 4, 5, 6, 7, 8, 9, 10] (0 + 1)
f [3, 4, 5, 6, 7, 8, 9, 10] ((0 + 1) + 1)
f [4, 5, 6, 7, 8, 9, 10] (((0 + 1) + 1) + 1)
f [5, 6, 7, 8, 9, 10] ((((0 + 1) + 1) + 1) + 1)
 ...
f [10] (((((((((0 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1)
f [] ((((((((((0 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1)
```

Isommilla listoilla lopputuloksena on massiivinen laskematon lauseke, joka vie turhaan tietokoneen muistia. Pahimmassa tapauksessa ohjelmoija voi huomaamattaan tehdä funktion, joka kasaa laskematonta lauseketta koko ohjelman suorituksen ajan, ja ohjelman käyttämä muistin määrä kasvaa jatkuvasti. GHC osaa parhaimmilla optimointiasetuksilla estää listauksessa 3.1 esiintyvän tilavuodon, mutta tilavuodon estoa ei kannata jättää pelkän kääntäjän optimoinnin varaan, vaan estää itse laiskuus pakottamalla ongelmallisten lausekkeiden laskeminen (Yang 2011).

4 Reaktiivinen funktio-ohjelmointi

Tässä luvussa kerrotaan reaktiivisesta funktio-ohjelmoinnista ja sen hyödyntämisestä peliohjelmoinnissa. Tässä luvussa kerrotaan myös nuolien käytöstä, koska niillä voi paikata tilavuotoja reaktiivisessa funktio-ohjelmoinnissa.

4.1 Peliohjelmointi reaktiivisella funktio-ohjelmoinnilla

Courtney, Nilsson & Peterson (2003) käyttivät Space Invaders peliä tehdessä reaktiiviseen funktio-ohjelmointiin tarkoitettua Yampa-kirjastoa. Yampa on Haskell-ohjelmointikielen sisälle rakennettu täsmäkieli (engl. *domain-specific language*), joka on tehty käyttäen nuolia. Nuolista on kerrottu lisää luvussa 4.2. Yampa-kirjastoa on käytetty pelinkehityksen lisäksi myös ainakin robotiikassa, äänen syntetisoinnissa ja käyttöliittymissä (Liu, Cheng & Hudak 2009). Reaktiivinen funktio-ohjelmointi tuo uudeksi elementiksi ajan kulun puhtaasti funktionaaliseen ohjelmointityyliin. Yhdenmukainen ajan kulun määrittely tukee määrittelevää ohjelmointityyliä (engl. *declarative programming*). Yampa-kirjaston alla olevan monimutkaisen toiminnan piilotus parantaa ohjelman selkeyttä ja lähdekoodin luettavuutta. (Courtney ym. 2003.)

Reaktiivinen funktio-ohjelmointi sopii peliohjelmointiin hyvin, koska monia peleissä olevia asioita voidaan mallintaa signaaleina. Esimerkiksi näppäimistön yksittäistä näppäintä voisi ajatella jatkuvana signaalina, jonka arvo riippuu siitä, että onko näppäin painettuna vai ei tietyllä ajanhetkellä. Jonkin pelissä olevan kappaleen sijaintia voi ajatella myös jatkuvana signaalina, joka riippuu muista signaaleista ja tapahtumista.

4.2 Signaalit ja nuolet

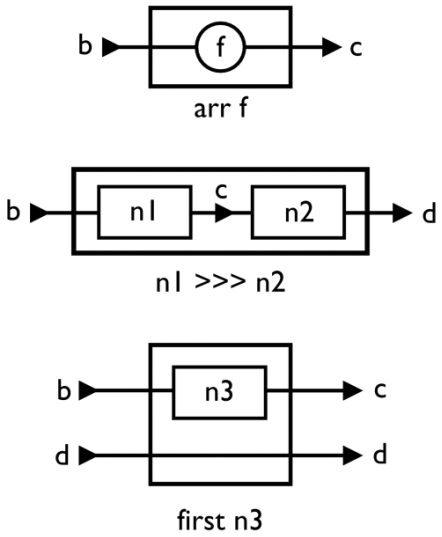
Nuolet ovat monadien yleistyksiä ja ne kuvaavat laskentaa abstraktilla tasolla (Liu, Cheng & Hudak 2009). Nuolilla voi vähentää huomattavasti vahingossa tehtyjen tilavuotojen määrää reaktiivisella funktio-ohjelmoinnilla tehdyissä ohjelmissa (Liu

ja Hudak 2007). Nuolilla pystyy tekemään ohjelmista modulaarisempia, ja niiden toimintaa on helpompi päätellä.

Haskelissa nuolia voidaan kuvata listauksessa 4.1 olevalla tyyppiluokalla. Kuviossa 3 on visualisoitu listauksessa 4.1 olevan `Arrow`-tyyppiluokan funktioita. Laatikot vastaavat nuolia ja ympyrät ovat funktioita. `arr` korottaa funktion "puhtaaksi" nuolella kuvatuksi laskennaksi, eli sen ulostulo riippuu täysin syötteestä. $(>>>)$ funktio yhdistää kaksi nuolta yhdistämällä ensimmäisen nuolen ulostulon toisen nuolen sisääntuloon. Tämä vastaa `Monad`-tyyppiluokan funktiota `(»=)`. `first` tekee nuolen, jossa laskentaa suoritetaan vain osalle syötteestä, loput syötteestä jatkaa eteenpäin muuttumattomana. (Liu & Hudak 2007.)

Listaus 4.1: Arrow-tyyppiluokka (Liu ym. 2009)

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



Kuvio 3: Arrow-tyyppiluokan funktioiden visualisointi

Reaktiivisen funktio-ohjelmoinnissa ohjelmointi tapahtuu signaalien tasolla. Signaalit ovat jatkuvia arvoja, jotka muuttuvat ajan suhteen. Signaaleita voi yhdistää ja nii-

tä voi integroida ja derivoida. (Liu & Hudak 2007.)

Käytännössä signaalien toteuttamiseen on kaksi eri tapaa. Ensimmäisessä tavassa signaaleita ajatellaan virtoina, jotka ovat käytännössä funktioita, joille annetaan lista diskreettejä aikoja ja ne palauttavat listan signaalin arvoja. Toisessa tavassa signaalit ovat pareja[1], jotka muodostavat jatkumon (engl. *continuation*). Pari sisältää signaalin arvon ja funktion, joka ottaa ajan muutoksen ja palauttaa tulevaisuuden parin. (Liu & Hudak 2007.) Parin sisältämä funktio palauttaa uuden tulevaisuuden parin, joka sisältää myös funktion, joka palauttaa taas uuden tulevaisuuden parin ja tästä muodostuu loputon jatkumo.

[1]Pari on tietorakenne, joka sisältää kaksi arvoa.

5 Yhteenveto

Pelialalla käytetään paljon imperatiivisia ohjelmointikieliä, joista käytetyin kieli on C++. Funktio-ohjelmoinnin hyödyntämistä peliohjelmoinnissa ei ole tutkittu paljoa.

Funktio-ohjelmoinnin käyttö tarjoaa monia etuja ohjelmointiin. Funktionaalisilla ohjelmointikielillä on helpompaa tehdä ohjelmia, joissa käytetään yhtäaikaista laskentaa tai rinnakkaislaskentaa. Puhtaiden funktioiden käyttö vähentää ohjelmassa olevia bugeja ja auttaa päättelemään ohjelman toimintaa ja suorituksen kulkua.

Funktionaaliset ohjelmointikielet, joissa on mahdollisuus laiskaan laskentaan, antavat mahdollisuuden tehdä vahingossa tilavuotoja. Tilavuodot aiheuttavat turhaa muistinkäyttöä. Pahimmassa tapauksessa muistin käyttö voi kasvaa jatkuvasti ohjelman koko suorituksen ajan. Reaktiivisen funktio-ohjelmoinnin käyttö nuolien kanssa vähentää tilavuotoja kuitenkin merkittävästi.

Tässä kandidaatintutkielmassa tehtiin kirjallisuuskatsaus, jonka tavoitteena oli selvittää mitä hyötyjä ja haittoja funktio-ohjelmointi voi tuoda peliohjelmointiin. Tutkielmassa käsiteltiin lyhyesti myös funktio-ohjelmoinnin perusteita. Tutkielmassa pohdittiin funktio-ohjelmoinnin etuja vain teoreettisella tasolla. Funktio-ohjelmoinnin ja erityisesti reaktiivisen funktio-ohjelmoinnin soveltuvuutta pelien tekoon pitäisi tutkia käytännönläheisemmin, esimerkiksi toteuttamalla jokin peli käyttäen reaktiivista funktio-ohjelmointia.

Kirjallisuutta

Courtney, A., Nilsson, H. & Peterson, J. 2003. *The Yampa arcade*. Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, s. 7–18.

Gershenfeld, A., Loparco, M. & Barajas, C. 2003. *Game plan: the insider's guide to breaking in and succeeding in the computer and video game business*. St. Martin's Griffin Press, New York.

Harris, T., Marlow, S., Peyton Jones, S. & Herlihy, M. 2008. *Composable memory transactions*. Communications of the ACM - Designing games with a purpose, Volume 51, Issue 8, s. 91–100.

Hinsen, K. 2009. *The Promises of Functional Programming*. Computing in Science & Engineering, Volume 11, Issue 4, s. 86–90.

Hughes, J. 1989. *Why functional programming matters*. The Computer Journal, Volume 32, Issue 2, s. 98–107.

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. 2007. *A history of Haskell: being lazy with class*. Proceedings of the third ACM SIGPLAN conference on History of programming languages, s. 12-1–12-55.

Jones, C. 1995. *Patterns of large software systems: failure and success*. IEEE Computer, Volume 28, Issue 3, s. 86–87.

Liu, H., Cheng, E. & Hudak, P. 2009. *Causal commutative arrows and their optimization*. Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, s. 35–46.

Liu, H. & Hudak, P. 2007. *Plugging a Space Leak with an Arrow*. Electronic Notes in Theoretical Computer Science (ENTCS) archive, Volume 193, s. 29–45.

McGill, M. 2008. *Critical skills for game developers: an analysis of skills sought by industry*. Proceedings of the 2008 Conference on Future Play: Research, Play, Share, s. 89–96.

Ted Tschang, F. 2005. *Videogames as interactive experimental products and their manner of development*. International Journal of Innovation Management, Vol. 9, No. 1, s. 103–131.

Tibell, J. 2010. *High-Performance Haskell*. Proceeding CUFP'10 ACM SIGPLAN Commercial Users of Functional Programming, Article No. 3. Esityksen diat saata-

villa WWW-muodossa <URL: http://johantibell.com/files/slides.pdf>. Viitattu 13.4.2015.

Wadler, P. 1997. *How to declare an imperative*. ACM Computing Surveys (CSUR), Volume 29, Issue 3, s. 240–263.

Yang, E. 2011. *Anatomy of a thunk leak*. Saatavilla WWW-muodossa <URL: http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/>. Viitattu 16.2.2015.