Aalto-yliopisto
Perustieteiden korkeakoulu
Informaatioverkostojen koulutusohjelma

Laiska evaluointi funktionaalisessa ohjelmoinnissa

Kandidaatintyö, keskeneräinen luonnos

21. helmikuuta 2016

Atte Keinänen

Aalto-yliopisto
Perustieteiden korkeakoulu
Informaatioverkostojen koulutusohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä:	Atte Keinänen
Työn nimi:	Laiska evaluointi funktionaalisessa ohjelmoinnissa
Päiväys:	21. helmikuuta 2016
Sivumäärä:	Täydennetään myöhemmin
Pääaine:	Informaatioverkostot
Koodi:	SCI3026
Vastuuopettaja:	Professori Juho Rousu
Työn ohjaaja(t):	Yliopistonlehtori Juha Sorva (Tietotekniikan laitos)
Kirjoitan tiivistelmän vasta viimeisenä.	
Avainsanat:	laiska evaluointi, call-by-need, evaluointistrategia, suoritusjärjestys, funktionaalinen ohjelmointi, Haskel
Kieli:	Suomi

Sisältö

1 Johdanto

Laiska evaluointi on 1970-luvulta juontuva ohjelmoinnin konsepti, jonka ytimessä on turhan ja liian aikaisen laskennan välttäminen ohjelmakoodia suorittaessa. Eli jos esimerkiksi listan alkiota ei koskaan tarvita, niin sen arvoakaan ei ole syytä laskea. Vastaavasti jos arvoa tarvitaankin, sitä ei lasketa listaa muodostettaessa, vaan vasta sitten kuin sitä on tarve käyttää.

Laiska evaluointi on nimityksenä vakiintunut, mutta "laiskuuden" osalta hieman harhaanjohtava. Paremmin konseptia kuvaa sen englanninkielinen synonyymi *call-by-need evaluation*, joka korostaa sitä, että laskentaa tehdään vasta kun tietoa tarvitaan. Toinen tulkinta on, että laskentaa viivytetään niin kauan, kunnes arvo on pakko laskea.

Laiskan evaluoinnin tekee aiheena kiinnostavaksi se, että sitä hyödynnetään kasvavissa määrin niin ohjelmointikielien suunnittelussa kuin yksittäisissä sovelluskirjastoissa. Esimerkiksi uudehkot ohjelmointikielet, kuten Scala ja Clojure, tarjoavat erityisiä syntaktisia rakenteita sitä varten. Toisaalta myös monet suositut apukirjastot, esimerkiksi JavaScript-kielen Immutable.js, Lazy.js ja Bacon.js, hyödyntävät sen periaatteita.

Laiskaa evaluointia ei ole yhtä tapaa tehdä "oikein", vaan käyttökohteiden ja toteutustapojen diversiteetti on suuri. Myös tavoiteltavat hyödyt ovat erilaisia. Verrataan esimerkiksi ohjelmointikielen ja apukirjaston laatijoiden tavoitteita: ohjelmointikielen laatija voi käyttää laiskaa evaluointia kielen ilmaisuvoiman kasvattamiseen, kun taas apukirjaston laatija voi päästä sen avulla kilpailevia kirjastoja parempaan suorituskykyyn.

Ymmärrän tässä työssä laiskan evaluoinnin laajasti, ja pyrin luomaan kokonaiskuvaa konseptin käyttökohteista ja toteutustavoista. Seuraavaksi määrittelen tarkemmin, mitä tarkoitan laiskalla evaluoinnilla, ja mitä muita käsitteitä konseptiin tiiviisti liittyy. Sen jälkeen esittelen tutkimuskysymykset, joihin työni pyrkii vastaamaan.

1.1 Keskeisten käsitteiden määrittely

Lauseke tarkoittaa ohjelmointikielen ilmaisua, jolle voidaan määrittää arvo. Esimerkiksi aritmeettiset operaatiot, muuttujien nimet ja funktiokutsut ovat lausekkeita.

Evaluointi tarkoittaa tarkoittaa lausekkeen arvon laskemista.

Evaluointistrategia tarkoittaa ohjelmointikielen sääntöjä, jotka määrittävät, missä tilanteissa lausekkeita evaluoidaan.

Laiska evaluointi tarkoittaa evaluointistrategiaa, joka viivyttää lausekkeen evaluointia saakka, kunnes sitä tarvitaan (Watt, 2004). Käsitettä voi käyttää kuvaamaan myös yksittäistä lauseketta abstraktimman operaation, kuten esimerkiksi kokoelmien käsittelyn, suorittamisen viivästämistä vastaavaan tapaan.

1.2 Tutkimuskysymykset

Haluan saada kartoitettua työlläni laiskan evaluoinnin käyttökelpoisuutta tutkimuksessa ja työelämässä, ja haluan myös tuottaa sellaista tietoa, joka olisi käyttökelpoista laiskan evaluoinnin konseptien yliopisto-opetuksen suunnittelutyössä. Päädyin näiden tavoitteiden kautta seuraaviin tutkimuskysymyksiin:

1. Mikä on laiskan evaluoinnin merkitys nykypäivänä?
2. Mitkä ovat laiskan evaluoinnin hyödyt ja haitat?

Kysymyksessä 1 laiskan evaluoinnin merkityksen tutkiminen tarkoittaa, että tarkastelen (a) laiskan evaluoinnin varhaista historiaa pohjustuksenomaisesti, (b) laiskan evaluoinnin leviämistä moderneihin ohjelmointikieliin sekä kielien apukirjastoihin, (c) laiskan evaluoinnin kautta kehittyneitä uusia ohjelmoinnin konsepteja ja (d) laiskaa evaluointia käyttävien teknologioiden hyödyntämistä tutkimuksessa ja työelämässä.

Kysymyksessä 2 selvitän sitä, millaisia subjektiivisia mielipiteitä laiskaa evaluointia hyödyntäviä teknologioita käyttävillä ihmisillä on sekä laiskasta evaluoinnista yleisesti, että kysymyksen 1 tarkastelussa esiin tulleista ohjelmointikielistä ja apukirjastoista.

1.3 Tutkimusmenetelmät

Tarkastelen kysymystä 1 narratiivisen kirjallisuuskatsauksella avulla. Tavoitteena on kartoittaa, millaista tietoa aiheesta tällä hetkellä on, ja tuoda sitä yhteen helppotajuisen narratiivin muotoon. Tällä tavoin luotu katsaus soveltuu hyvin materiaaliksi opettajille (Baumeister ja Leary, 1997, s. 312), joten menetelmävalinta on linjassa sen tavoitteen kanssa, että työstä olisi hyötyä yliopisto-opetuksen suunnittelussa.

Kirjallisuuskatsauksen materiaalina käytän ensisijaisesti akateemisia artikkeleita, joita aiheesta on kirjoitettu paljon. Hain artikkeleita ensisijaisesti Scopus-tietokannasta, jossa käytin seuraavaa hakulauseketta:

```
("lazy evaluation" OR "non-strict" OR "call-by-need" OR "call by need")
AND ("functional")
```

And-operaattorin vasemmalla puolella on kaikki laiskan evaluoinnin tyypillisimmät synonyymit vaihtoehtoisina hakutermeinä.[1] Operaattorin oikealla puolella oleva hakutermi *functional* rajaa hakutulokset funktionaalisen ohjelmoinnin piiriin. Tämä hakulauseke

[1]Puhtaan funktionaalisen ohjelmoinnin, erityisesti Haskell-ohjelmointikielen, sanastossa käsitteellä *non-strict* on usein laiskasta evaluoinnista eriävä merkitys. Tätä käsitellään tarkemmin luvussa 2. Kirjallisuudessa käsitteet kuitenkin esiintyvät myös toistensa synonyymeina.

tuotti otsikosta, tiivistelmästä ja avainsanoista haettaessa 326 tulosta. Tämä oli sopiva määrä tuloksia tutkimuskysymysten kannalta relevanteimpien artikkeleiden valikointia ajatellen.

Scopus-tietokannan lisäksi seuloin artikkeleita samalla hakutermillä myös Google Scholarista. Lisäksi Stack Overflow -kysymyspalvelun vastausten ja Haskellin oman wiki-alustan kautta löytyi useita linkkejä relevantteihin artikkeleihin. Täydensin akateemisista artikkeleista saatua tietoa myös vähäisissä määrin blogikirjoituksilla. Niistä on iloa sellaisten uusien kehityskulkujen esittelyssä, joiden esiintyminen akateemisissa artikkeleissa on hyvin vähäistä.

Kysymykseen 2 vastatakseni käytän yhdistelmää eri tiedonhakutapoja. Hyödynnän (a) kirjallisuuskatsauksen tiedonhaussa vastaan tulleita mielipidelatautuneita artikkeleita, (b) blogikirjoituksia ja (c) kysymällä mielipiteitä työkavereiltani Slack-kommunikaatioalustan avulla. Blogikirjoituksissa painotan kirjoittajia, joilla on myös akateemista näyttöä aihepiiristä. Mielipiteen kysymisessä pyydän vastaajia kertomaan, että mikäli he ovat missään kontekstissa käyttäneet laiskaa evaluointia esimerkiksi ohjelmointikielessä tai apukirjastossa, millaisia vahvuuksia tai heikkouksia he ovat kohdanneet.

Kirjallisuuskatsauksen ja mielipideselvityksen käsittelyjärjestystä tässä työssä esittelee seuraava osio, työn rakenne.

1.4 Työn rakenne

Tutkimuskysymykseen 1 vastausta pohjustan luvussa 2, joka käsittelee laiskan evaluoinnin historiaa ja keskeisempiä konsepteja. Se antaa lukijalle valmiudet ymmärtää laiskan evaluoinnin nykyisiä sovellutuksia, jotka ovat luvun 3 aiheena. Luvussa 4 käsittelen subjektiivisesti koettuja etuja ja haittoja, joita laiskaan evaluointiin yleensä ja laiskan evaluoinnin sovellutuksiin liittyy. Luvussa 5 vedän työtä yhteen ja tarkastelen työn rajausta.

2 Varhainen historia ja keskeisimmät konseptit

Laiska evaluointi sai alkusysäyksensä 1970-luvulla. Sarja julkaisuja loi pohjaa ajatukselle laiskoista funktionaalisista kielistä työkaluna käytännönläheiseen ohjelmistokehitykseen. Ajatus esiteltiin ensimmäisenä matemaattisesti lamdakalkyylin, funktionaalisen ohjelmoinnin kannalta keskeisen matemaattisen teorian, näkökulmasta (Wadsworth, 1971). Viisi vuotta myöhemmin julkaistiin toisistaan riippumatta kolme artikkelia (Henderson ja Morris Jr, 1976; Friedman, 1976; Turner, 1976), joissa esiteltiin laiskaa evaluointia ohjelmoinnin perspektiivistä.

1980-luvulla vaihteessa oli uraauurtavaa kehitystä kohti ensimmäisiä laiskoja ohjelmointi-

kieliä. Saman aikaan kehitettiin myös täysin uudenlaisia tietokoneita, jotka kilpailivat alan standardin, Von Neumannin arkkitehtuurin, kanssa. Kuitenkin pidemmän päälle osoittautui, ettei tarvetta erikoisille arkkitehtuureille ole, vaan hyvin laadituilla ohjelmointikielten kääntäjillä voidaan päästä hyviin lopputuloksiin myös Von Neuman -arkkitehtuuria hyödyntävissä tietokoneissa. (Hudak et al., 2007)

2.1 Ei-tiukka semantiikka ja graafireduktio

Yhdistävää 1980-luvun vaihteessa kehitetyille laiskoille ohjelmointikielille oli, että niissä alettiin hyödyntämään sittemmin vakiintuneita *ei-tiukan semantiikan* ja *graafireduktion* periaatteita. Jos ohjelmointikieli perustuu ei-tiukalle semantiikalle, niin lausekkeella voi olla arvo, vaikka jollakin sen alilausekkeista (pienemmistä lausekkeista, joista lauseke koostuu) ei olisi arvoa. Vastaavasti *tiukkaan semantiikkaan* perustuvat ohjelmointikielet toimivat siten, että jos joltakin alilausekkeelta puuttuu arvo, niin lausekkeellakaan ei ole arvoa, ja tällöin lausekkeen evaluoinnin voi ajatella epäonnistuneen (Scott, 2009, s. 523).

Tiukkaa semantiikkaa käytetään yleisesti imperatiivisissa ohjelmointikielissä. Useimmissa suosituissa imperatiivisissa kielissä, esimerkiksi Pythonissa, tiukan semantiikan toteutus perustuu funktioiden evaluointijärjestykseen liittyviin sääntöihin. Evaluointi etenee "sisimmät ensin" -periaatteella. Siinä funktion kaikki argumentit evaluoidaan ennen kuin funktiota kutsutaan, ja evaluointi etenee sisimmästä funktiokutsusta ulompia funktiokutsuja kohti.

Vastaavasti ei-tiukan semantiikan tyypillisin toteutus, jota tapaa esimerkiksi Haskellissa, perustuu "uloimmat ensin" -periaatteeseen. Siinä funktioiden evaluointi etenee uloimmasta funktiokutsusta kohti sisempiä, ja ainoastaan ne funktion argumentit, joita funktio todellisuudessa käyttää, evaluoidaan. Siten Haskellissa on mahdollista tehdä jopa sellaisia funktioita, joka ei tarvitse argumentteja lainkaan, ja siten palauttaa arvon riippumatta siitä, onko parametrilausekkeilla arvoa vai ei. (HaskellWiki, 2013).

Taulukko 1 demonstroi semantiikan vaikutusta funktioiden evaluointiin. Siinä on toiminallisuudeltaan vastaava koodi suoritetaan sekä Pythonilla että Haskellilla. Funktio `noreturn` aiheuttaa ikuisen silmukan, minkä tähden se ei koskaan palauta arvoa. Haskellilla lausekkeen evaluointi palauttaa arvon, koska `noreturn` -funktiota ei koskaan kutsuta.

Python, tiukka semantiikka	Haskell, ei-tiukka semantiikka

`def noreturn(x):`
`  while True:`
`  x = -x`
`  return x # not reached`
 	
`def even(x):`
`  return x % 2 == 0`
 	
`> any(even(n) for n in [3, 2, noreturn(6)])`
`⇒ (ei palauta arvoa)`
	`noreturn :: Integer -> Integer`
	`noreturn x = negate (noreturn x)`
	
	`> any . even . [3, 2, noreturn 6]`
	`⇒ True`
	
	Katsotaan, haluanko tässä vielä käydä evaluoinnin välivaiheita läpi. Onko tarpeellista?

Taulukko 1: Esimerkki kielen semantiikan vaikutuksesta evaluointijärjestykseen

Graafireduktio on tapa ei-tiukan semantiikan toteuttamiseen. Se esittää lausekkeet verkon muodossa, mikä mahdollistaa toistuvien lausekkeiden jakamisen muiden lausekkeiden kesken (Hudak, 1989). Esimerkiksi lausekkeen `(1+2)*(1+2)` verkkoesityksessä lauseke `(1+2)` pystytään jakamaan, minkä myötä sen arvo tarvitsee evaluoida vain kerran.

Ei-tiukan semantiikan "uloimmat ensin" -suoritusjärjestys ja graafireduktiossa tapahtuva lausekkeiden jakaminen luovat perustan laiskalle evaluoinnille. Lausekkeen arvoa ei lasketa ennen kuin on tarve, eikä sitä turhaan lasketa uudelleen, jos sitä käytetään myöhemmin uudelleen.

2.2 Haskellin ja puhtaan funktionaalisen ohjelmoinnin kehitys

Hudak et al. (2007) kuvaa, kuinka 1980-luvun puolivälissä laiskaa evaluointia hyödyntäviä ohjelmointikieliä alkoi olla ruuhkaksi asti. Useimmat kielistä soveltuivat vain kapeaan määrään käyttökohteita, ja niillä ei ollut riittävää määrää käyttäjiä suuren suosion saavuttamiseksi. Kuitenkin artikkelin kirjoittajat olivat tällöin sitä mieltä, että kielet muistuttivat ominaisuuksiltaan hyvin paljon toisiaan. Alkoi kehittyä ajatus siitä, että olisi hyvä luoda yksi, yleinen kieli, joka korvaisi kerralla monia aikaisempia kieliä.

Tämä johti Haskell-ohjelmointikielen kehityksen aloittamiseen. Siitä vastasi Haskell-komitea, jossa vaikutti monia aikaisempien laiskaa evaluointia hyödyntäneiden kielien suunnittelijoita. Komitea onnistui keräämään yhteen aikaisemmin erillään samaa aihepiiriä tutkineita, mikä kiihdytti laiskan evaluoinnin parissa tapahtunutta tieteellistä kehitystä. Alkuvaiheen kehitystyö oli onnistunutta, ja Haskellista kehittyi etenkin tietojenkäsittelytieteen akateemisen tutkimuksen parissa suosittu kieli.

Haskell ei määritelmällisesti ole laiska ohjelmointikieli, vaan jo Haskell 1.1 -raportissa (Hudak et al., 1991) se määriteltiin ainoastaan ei-tiukkaa semantiikkaa käyttäväksi kieleksi. Käytännössäkään kieli ei ole täysin laiska, vaan Haskellin kääntäjät tekevät useita nopeusoptimointeja suorittamalla tiettyjä osia koodista tiukkaan semantiikan säännöllä. Useimmat kielen rakenteet noudattavat kuitenkin oletusarvoisesti laiskan evaluoinnin periaatteita.

Laiskassa evaluoinnissa koodia ei suoriteta imperatiivisten kielien tapaan rivi riviltä, vaan suoritusjärjestys määräytyy tarpeen mukaan. IO-operaatioita, eli esimerkiksi näytölle tulostamista tai käyttäjän syötteen odottamista, ei pystytä suorittamaan halutussa järjestyksessä pelkkien funktiokutsujen avulla. Tästä seurasi, että Haskellista tuli *puhtaasti funktionaalinen* ohjelmointikieli. Käsitteellä tarkoitetaan, että kielen funktiokonstruktiot ovat funktioita matemaattisessa mielessä: funktion kutsuminen samoilla argumenteilla palauttaa aina saman arvon, eikä funktioilla voi suorittaa IO-operaatioita tai muita ohjelman tilaa muuttavia operaatioita.

Laiskan evaluoinnin periaatteiden noudattaminen on johtanut moniin innovaatioihin funktionaalisen ohjelmoinnin saralla. Koska Haskell on puhtaasti funktionaalinen kieli, täytyi kehittää kokonaan uusia konsepteja, jotta esimerkiksi IO-operaatioiden suorittaminen halutussa järjestyksessä ja niiden ketjuttaminen toisiinsa onnistuu. Tunnetuimmaksi konseptiksi on noussut *monadi*. Alkuperäisessä monadin konseptin esitelleessä artikkelissa sitä kutsutaan "imperatiivisen funktionaalisen ohjelmoinnin" työkaluksi, mikä kuvaa monadien mahdollistamia sekventiaalisia IO-operaatioita (Peyton Jones ja Wadler, 1993).

Niin monadi kuin monet muutkin Haskelliin kehitetyt puhtaan funktionaalisen ohjelmoinnin konseptit ovat levinneet myös sellaisiin ohjelmointikieliin, jotka ovat evaluointistrategialtaan tiukkoja. Siten laiskan evaluoinnin vaalimisen Haskellissa voi ajatella olleen hyödyksi tietojenkäsittelytieteen kehitykselle laajemminkin.

Mainintaa tähän osioon vielä (1) thunkeista eli evaluoimattomista arvoista liittyen Haskellin evaluointiin ja (2) pysyvätilaisista muuttujista liittyen Haskellin puhtaasti funktionaaliseen luonteeseen?

2.3 Päättymättömät tietorakenteet ja kontrollirakenteet

Jo 1980-luvun varhaisessa tutkimuksessa käsiteltiin laiskan evaluoinnin mahdollistamia laiskoja tietorakenteita, etenkin "päättymättömiä tietorakenteita". Näitä ovat muun muassa päättymättömät listat, puurakenteet ja tapahtumavirrat. Ne ovat tyypillisesti rekursion avulla toteutettuja, ja laiskan evaluoinnin ansiosta tietorakenteessa tarvitsee mennä vain niin "syvälle" kuin on tarvetta.

Eräs klassinen esimerkki on matematiikan äärettömiä lukujonojen kuvaaminen. Seuraavassa on Fibonaccin lukujono ilmaistuna rekursiivisesti Haskellilla. Huomionarvoista on, kuinka paljon koodi muistuttaa tapaa, jolla rekursiivinen lukujono ilmaistaisiin matematiikassa:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Laiska evaluointi mahdollistaa myös omien kontrollirakenteiden kirjoittamisen. Esimerkiksi if-ehtolauseesta on Haskellissa helppoa tehdä oma versio, jossa ehdon täyttymisen perusteella evaluoidaan vain jompikumpi vaihtoehtoisista lausekkeista:

```
% Käyttö: myIf condition onTrue onFalse
myIf :: Bool -> a -> a -> a
myIf True  x _ = x
myIf False _ y = y
```

Omien kontrollirakenteiden tuki helpottaa kieleen upotettujen *täsmäkielten* rakentamista. Täsmäkielillä tarkoitetaan pieniä, tavanomaisesti deklaratiivisia kieliä, jotka ovat hyvin ilmaisuvoimaisia tietyssä ongelma-avaruudessa (Van Deursen et al., 2000). Täsmäkielet toteutetaan usein ohjelmointikielen apukirjastoina, ja täsmäkieli on siinä tapauksessa yhdistelmä ohjelmointikielen valmiita ominaisuuksia ja apukirjastossa siihen laadittuja lisäyksiä.

Seuraavassa luvussa siirryn tarkastelemaan sitä, millaisia sovellutuksia laiskalle evaluoinnille on löydetty erilaisissa ohjelmointikielissä ja ohjelmointiin liittyvissä konteksteissa.

3 Sovellutuksia ohjelmointikielissä ja apukirjastoissa

Laiskalle evaluoinnille on löydetty lukuisia erilaisia sovellutuksia. Tähän lukuun on koottu niistä merkittävimpiä. *Kirjoitan tämän pohjustustekstin uudelleen, kun alaluvut ovat sisällöltään jokseenkin lopullisessa kuosissa.*

3.1 Laiskat muuttujat ja by-name-parametrit Scalassa

Scala on sekä funktionaalista että imperatiivista ohjelmointia tukeva yleiskäyttöinen ohjelmointikieli. Scala-lähdekoodi kääntyy Java-tavukoodiksi, ja kieli on täysin yhteensopiva

Java-kirjastojen kanssa. Scala on semantiikaltaan tiukka kieli, mutta evaluointijärjestystä on mahdollista muuttaa paikallisesti kahdella mekanismilla: *laiskoilla muuttujilla* ja *by-name-parametreilla*.

Laiska muuttuja luodaan lisäämällä `lazy` -avainsana muuttujan määrittelyn alkuun. Kyseisen muuttujan arvon määrittävä lauseke evaluoidaan vasta sitten, kun muuttujaa käytetään ensimmäistä kertaa. Arvo myös sidotaan muuttujaan, eli jos muuttujaa tarvitaan uudelleen, niin arvoa ei tarvitse evaluoida uudelleen.

By-name-parametri on funktion parametri, joka luodaan funktion määrittelyssä syntaksilla `parameterName: => Type`. Funktiokutsussa parametrin arvo evaluoidaan vasta, kun parametria käytetään. Laiskasta muuttujasta poiketen arvoa ei kuitenkaan evaluointihetkellä sidota parametriin, vaan jos parametria käytetään uudelleen, niin arvokin evaluoidaan uudelleen.

Laiskojen muuttujien ja by-name-parametrien avulla Scalassa pystyy toteuttamaan monia laiskalle evaluoinnille ja puhtaalle funktionaaliselle ohjelmoinnille tyypillisiä piirteitä, kuten omia kontrollirakenteita ja laiskoja listoja Chiusano ja Bjarnason (2014). Siten Scalaa pidetään myös tehokkaana kielenä täsmäkielien toteuttamiseen.

Voisi lisätä vielä asiaa ei-tiukoista transformereista ja streameista eli laiskoista sekvensseistä. Nämä voisivat tulla luvun loppuun, jolloin siirryttäisiin loogisesti matalan tason rakenteista (mm. laiskat muuttujat) korkeammalle tasolle (mm. laiskat listat).

3.2 Laiskat sekvenssit Clojuressa

Clojure on funktionaalinen, Lisp-kieleä muistuttava yleiskäyttöinen ohjelmointikieli, joka Scalan tapaan kääntyy Java-tavukoodiksi ja on täysin yhteensopiva Java-kirjastojen kanssa. Clojure on semantiikaltaan tiukka kieli, eikä se tue Scalan tapaan yleistetysti laiskoja muuttujia ja parametreja. Kieli kuitenkin tukee laiskoja sekvenssejä, eli listamaisia tietorakenteita, joita varten kielessä on valmis tuki `lazy-seq` -makron avulla.

Käytännössä Clojuren käyttäjä tulee käyttäneeksi laiskoja sekvenssejä paljon, sillä monet kielen yleisimmistä listamaisten tietorakenteiden operaatioista palauttavat laiskan sekvenssin. Näitä operaatioita ovat esimerkiksi `map`, `filter` ja `take`. Clojuren laiskojen listojen arvoja evaluoidaan sitä mukaan kuin niitä tarvitaan, ja arvot pysyvät muistissa tulevia käyttökertoja varten.

Clojure tukee *makroja*, joiden avulla kielen kääntäjää pystyy laajentamaan ohjelmakoodista käsin, ja sen myötä kieleen voi luoda omia kontrollirakenteita. Tämän myötä Clojure ei tarvitse laiskaa evaluointia ollakseen silti tehokas kieli täsmäkielien luomiseen.

3.3 Laiskat sekvenssit JavaScript-apukirjastoissa

JavaScript on ainoa verkkoselainten yleisesti tukema ohjelmointikieli, ja lisäksi JavaScriptiä käytetään runsaasti myös palvelinohjelmointiin. Se on semantiikaltaan tiukka kieli, ja tukee rajoitetusti funktionaalisen ohjelmoinnin konsepteja. JavaScriptiin on 2010-luvulla kehitetty useita apukirjastoja, jotka tuovat funktionaalisen ohjelmoinnin työkaluvalikoimaa laajemmin ohjelmoijien käyttöön. Näistä uusimmissa näkyy kasvavissa määrin laiskan evaluoinnin konseptien vaikutus kirjastosuunnitteluun.

Immutable.js on Facebookin julkaisema JavaScript-apukirjasto, joka tarjoaa tuen funktionaaliselle ohjelmoinnille ominaisille pysyvätilaisille tietorakenteille. Kaikki kirjastossa määritellyt sekvenssit, esimerkiksi indeksoidun listan `List` tai pinon `Stack`, voi muuttaa laiskaksi `Seq`-konstruktorilla. Sekvenssille kohdistettavat operaatiot ovat tämän luokan metodeja, ja kun `Seq`-instanssille kutsuu näitä metodeja, ne palauttavat uuden laiskan sekvenssin. Ketjutetut operaatiot evaluoidaan vasta sitten, kun lopullista arvoa tarvitaan.

`Seq` tukee myös päättymättömiä sekvenssejä kirjaston tarjoamien `Range`- ja `Repeat` -konstruktoreiden sekä iteraattorifunktioiden avulla. `Seq` ei kuitenkaan Clojuren laiskojen listojen tapaan pidä evaluoituja arvoja muistissa.

Lazy.js on JavaScriptin-apukirjasto sekvenssien käsittelemiseen. Siinä Immutablen laiskaa sekvenssin luontia vastaa konstruktori `Sequence`. Lazy.js ei Immutablen tapaan esittele uusia tietorakenteita, vaan sitä voi käyttää yhdessä JavaScriptin sisäänrakennettujen listamaisten tietorakenteiden kanssa. Toisin kuin Immutable.js, se myös tukee joitain funktionaalisen reaktiivisen ohjelmoinnin konsepteja, joita käsittelen seuraavassa luvussa.

Sekä Immutable.js että Lazy.js lupaavat, että viivyttämällä sekvensseille kohdistettujen operaatioiden evaluointia ne ovat merkittävästi nopeampia kuin kilpailijansa, joissa tyypillisesti listan arvo evaluoidaan uudelleen jokaisen listaoperaation kutsun yhteydessä.

Halutaanko sekä Clojure-osioon että tähän lähdeviittaukset kielen/kirjastojen dokumentaatioihin?

3.4 FRP ja reaktiivinen ohjelmointi JavaScript-apukirjastoissa

Funktionaalinen reaktiivinen ohjelmointi, lyhyemmin FRP, on deklaratiivinen ohjelmointiparadigma, jolla voi käsitellä muuttuvatilaisia arvoja funktionaalisessa ohjelmoinnissa. FRP esittää muuttuvatilaisen arvon aikariippuvaisena “signaalina” . Näitä signaaleja voi transformoida ja yhdistellä joustavasti (Czaplicki, 2012).

Paradigma sai alkunsa Haskellin tutkimuksesta, kun Elliott ja Hudak (1997) julkaisivat kokeellisen täsmäkielen nimeltä *Fran* animaation kuvaamiseen funktionaalisessa ohjelmoinnissa. Fran pyrki pääsemään irti grafiikkaohjelmoinnin perinteisestä imperatiivisesta

ja diskreetistä luonteesta, ja korvasti sen jatkuvaan aikatarkasteluun perustuvalla mallilla. Fran rakennettiin Haskellilla, ja siten siinä käytettiin laiskaa evaluointia laajasti.

Reaktiivinen ohjelmointi tarkoittaa laajemmin ohjelmointiparadigmaa, joka keskittyy datavirtojen ja niiden muutosten käsittelyyn. FRP on inspiroinut monia 2010-luvun suosittuja reaktiivisia kirjastoja, kuten *React*, *RxJS* ja *Bacon.JS*, ja näiden lisäksi myös uusia ohjelmointikieliä on luotu reaktiivisten periaatteiden pohjalta, kuten Elm. Nämä perustuvat diskreettien datavirtojen käsittelyyn, eivät jatkuviin datavirtoihin, minkä takia ne eivät vastaa FRP:n akateemista määritelmää (Elliott, 2010).

Valtaosa suosituista kirjastoista ei hyödynnä laiskaa evaluointia, koska ne ovat laadittu tiukan semantiikan ohjelmointikielille. JavaScriptille laadittu Bacon.js -kirjasto, jossa diskreettejä datavirtoja käsitellään funktionaalisen ohjelmoinnin työkaluilla, käyttää laiskaa evaluointia valikoidusti datavirtojen käsittelyoperaatioiden ketjuttamiseen. Tämä toimii samantyyppisellä mekanismilla kuin JavaScript-kirjastot Immutable.js ja Lazy.js.

Myös Lazy.js tukee diskreettien tapahtumavirtojen käsittelyä tavalla, joka on hyvin samankaltainen reaktiivisten kirjastojen, kuten RxJS ja Bacon.JS, kanssa. Kuitenkin kirjaston reaktiiviset ominaisuudet on jätetty kokeelliselle asteelle.

3.5 Muita sovellutuksia

En vielä ole perehtynyt seuraaviin aiheisiin ja niihin liittyviin artikkeleihin riittävästi nähdäkseni, ovatko ne aiheen kannalta relevantteja. Palaan niihin myöhemmässä vaiheessa kirjoitusprosessia.

- Digitaalisen logiikan simulointi / piirisuunnittelu (Charlton et al., 1991)
- Persistentit ohjelmointikielet (Wevers, 2014)
- Logiikkapohjainen ohjelmointi (oma ohjelmointiparadigmansa) (Alpuente et al., 1997)
- Java 8 streamit, LINQ/C#

4 Subjektiiviset edut ja haitat

Tulen kirjoittamaan tähän etujen/haittojen kartoitusprosessista sekä siitä, millaisiin alalukuihin olen jakanut osion ja miksi. Voi myös vähän käsitellä sitä miksi Haskell dominoi keskustelua. Se vie laiskan evaluoinnin selvästi pisimmälle suosituista kielistä, ja siten sitä on hyvä käyttää yleisenä benchmarkina laiskaan evaluointiin liittyvässä keskustelussa, toki terveen kritiikin kera.

4.1 Ilmaisuvoima

Hughes (1989) sanoo, että laiskan evaluoinnin käyttö tekee käytännölliseksi jakaa sovelluskoodia *tuottajiin* ja *valitsimiin*. Tuottajat määrittävät suuren määrän mahdollisia vastauksia, ja valitsin valitsee niistä relevanteimmat. Tämä tarjoaa sellaisen tavan modularisoida sovelluskoodia, joka on laiskasti evaluoiduille puhtaasti funktionaalisille kielille uniikki. Hughes kuvaa, että kyseessä on kenties tehokkain tapa modularisoida koodia funktionaalisen ohjelmoijan työpakissa.

Karvonen (2016) puolestaan väittää, että laiskuus ei merkittävästi kasvata kielen ilmaisuvoimaa. Useat sellaiset kirjastot, joiden väitetään olevan toteutettavissa vain laiskasti evaluoidun kielen avulla, ovat hänen mukaansa helposti toteutettavissa ei-laiskalla kielellä simulomalla laiskuutta muilla keinoin niissä kohti kirjastoja, kun sille on tarve.

Päättymättömät listat Harper (2011) toteaa tietorakenteeksi, jonka toteuttamiseksi laiska evaluointi ei ole välttämätön. Hänen mukaansa laiskuudella ja päättymättömien listojen tyyppisillä tietorakenteilla ei ole selkeää yhteyttä toisiinsa. ML-ohjelmointikielessä, joka on tiukasti evaluoitu, samat saman konseptin toteuttaminen onnistuu myös vaivatta.

Sekä Karvonen että Harper (2011) sanovat, että laiskuuden tähden Haskellista puuttuu tuki induktiivisille tietotyypeille, joka löytyy esimerkiksi kilpailevasta ML-ohjelmointikielestä. Tämän myötä kielellä ei ole mahdollista esimerkiksi luoda tietotyyppiä luonnollisten lukujen kuvaamiseen, tai tyyppi luonnosllisten lukujen muodostamalle listalle.

4.2 Ajan- ja tilankäytön ennustettavuus

Ajankäytön ennustettavuudella tarkoitetaan sitä, kuinka tarkkaan ohjelmoija pystyy arvioimaan, miten ajallisesti ohjelman suoritus etenee, ja kuinka paljon aikaa mikäkin operaatio vie. Tilankäytön ennustettavuus kuvaa vastaavasti sitä, kuinka tarkasti ohjelman muistinkäytön pystyy arvioimaan ohjelman suorituksen edetessä.

Haskell-ohjelmaa tehdessään Sampson (2009) kohtasi tiiminsä kanssa ongelmia yllättävien tilavuotojen muodossa. Tilavuodot johtuivat siitä, että kielen evaluoimattomat lausekkeet pitävät jostakin syystä sovelluksen näkökulmasta jo vanhentunutta tietoa muistissa. Syiden hän epäili liittyvän koodin rinnakkaisuuden toteutuksessa käytettyihin abstraktioihin.

Karvonen (2016) sanoo, että funktioiden yhdistämiseen liittyvät mahdollisuudet eivät ole Haskellissa parempia, toisin kuin usein väitetään. Esimerkiksi listaoperaatioiden yhdistämisestä ei tule käytännön hyötyä siksi, että on vaikeaa ennakoida, kuinka listaoperaatiot tarkalleen käyttäytyvät laiskasti evaluoidussa kielessä. Tämän seurauksena niiden tilankäytön ennustettavuuskin on heikko.

Artikkelissa X [pitää vielä kaivaa, mikä oli kyseessä] todetaan, että Haskell soveltuu heikosti reaaliaikaiseen grafiikkaan, esimerkiksi peleihin, koska evaluoimattomien lausekkeiden evaluointi isona kertaoperaationa saattaa pysäyttää grafiikan ajaksi, jonka käyttäjä huomaa häiritsevänä nykimisenä.

4.3 Vianetsintä ja suoritusjärjestys

Toistuvasti esiin noussut kritiikki laiskaa evaluointia ja Haskellia kohtaan on, että koodin ajonaikaisten ongelmien selvittäminen on hankalaa. Näin kokee muun muassa Daniels et al. (2012), jotka ilmaisivat syyksi, että testausta paljon helpottavalle *funktiokutsupinojen seuraamiselle* on laiskoissa funktionaalisissa kielissä heikko tuki. He toisaalta viittasivat myös siihen, että Haskellin kehittyessä kutsupinojen seuraaminen saattaa helpottua.

Samaan tapaan Pop (2010) kokevat ajonaikaisten ongelmien löytämisen vaikeaksi funktiokutsupinojen seurannan puutteen vuoksi. Haskellin tarjoamat vaihtoehdot ongelmien etsimiseen olivat hyvin primitiivisiä, ja yhdessä laiskuuden kanssa se teki hänen mielestään ongelmien löytämisestään aloittelevalle Haskell-kehittäjälle paljon vaikeampaa verrattuna perinteisempään skriptikieleen.

Sampson (2009) kokee, että oli vaikeaa hahmottaa, missä tilanteessa mikäkin arvo evaluoidaan, eli milloin ohjelma tekee työtä ja milloin ei. Apukirjastot usein palauttavat evaluoimattomia arvoja, jotka muodostuvat isosta määrästä lausekkeita, ja niiden evaluointi tapahtuu vasta sitten, kun arvoja käytetään. Arvon käyttämisen kohtaa voi olla kuitenkaan kirjoittajien mielestä vaikeaa löytää. Tämä liittyy hänen mukaansa myös ajonaikaiseten virheiden paikantamisen vaikeuteen.

4.4 Käytön laajuus

Karvonen (2016) sanoo laiskan evaluoinnin olevan ominaisuus, josta on tietyissä tilanteissa hyötyä, mutta ne kannattaa rajata tarkkaa. Siten hän kannattaa tiukasti evaluoituja ohjelmointikieliä ja laiskan evaluoinnin konseptien käyttöä niissä harkitusti.

4.5 Koettu arvo

Sampson (2009) toteaa artikkelin lopussa, että hän ja tiimi ovat projektin jälkeen epävarmoja laiskan evaluoinnin tuomasta arvosta: joka kerta kun he tuntevat saaneensa siitä hyvän otteen uusia ongelmia ilmenee. Hän kokee, että aiheesta pitäisi vähintäänkin olla olemassa hyvä kirja, joka käsittelisi kaikkia laiskan evaluoinnin kanssa ilmeneviä ongelmia sekä auttaisi luomaan lukijalle hyvä intuitio laiskasti evaluoitujen systeemien käyttäytymisestä.

5 Yhteenveto ja johtopäätökset

Haluan suoda ajatuksia tuloksista vedettäville johtopäätöksille vasta sitten, kun tuloksia käsittelevät luvut ovat jokseenkin lopullisessa muodossa. Minkähäntyyppisiä johtopäätöksiä voin ylipäätään vetää? Mikä on yhteenvedon rooli?

Metodologisia puutteita:

- varsin rajallinen määrä mielipiteitä mielipidetutkimuksessa

- kandin pituus on melko lyhyt, ja siksi kirjallisuuskatsaus jää väistämättä pintapuoliseksi

Lähteet

Maria Alpuente, Moreno Falaschi, Pascual Julian ja German Vidal. Specialization of lazy functional logic programs. *ACM SIGPLAN Notices*, osa 32, sivut 151–162. ACM, 1997.

Roy F Baumeister ja Mark R Leary. Writing narrative literature reviews. *Review of general psychology*, 1(3):311, 1997.

Colin C Charlton, D Jackson ja Paul H Leng. Lazy simulation of digital logic. *Computer-Aided Design*, 23(7):506–513, 1991.

Paul Chiusano ja Rnar Bjarnason. *Functional programming in Scala*. Manning Publications Co., 2014.

Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

Noah M Daniels, Andrew Gallant ja Norman Ramsey. Experience report: Haskell in computational biology. *ACM SIGPLAN Notices*, osa 47, sivut 227–234. ACM, 2012.

Conal Elliott. Why program with continuous time?, 2010. URL `http://conal.net/blog/posts/why-program-with-continuous-time`. Viitattu 18.2.2016.

Conal Elliott ja Paul Hudak. Functional reactive animation. *ACM SIGPLAN Notices*, osa 32, sivut 263–273. ACM, 1997.

DP Friedman. Cons should not evaluate its arguments. 1976.

Robert Harper. The real point of laziness, 2011. URL `https://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/`. Viitattu 20.2.2016.

HaskellWiki. Non-strict semantics, 2013. URL `https://wiki.haskell.org/Non-strict_semantics`. Viitattu 11.2.2016.

Peter Henderson ja James H Morris Jr. A lazy evaluator. *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, sivut 95–103. ACM, 1976.

P Hudak, Simon Peyton Jones, P Wadler, B Boutel, J Fairbairn, J Fasel, MM Guzman, K Hammond, J Hughes et al. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language. Version 1.1*. Yale University, Department of Computer Science, 1991.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

Paul Hudak, John Hughes, Simon Peyton Jones ja Philip Wadler. A history of haskell: being lazy with class. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, sivut 12–1. ACM, 2007.

John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

Vesa Karvonen. Kommentit reaktor-yrityksen slack-keskustelutyökalussa, 2016. URL `https://gist.github.com/attrck/486ef667b5dd7e6e61f2`. Keskustelu tallennettu GitHub Gistiin.

Simon L. Peyton Jones ja Philip Wadler. Imperative functional programming. sivut 71–84, 1993.

Iustin Pop. Experience report: Haskell as a reagent. *ACM SIGPLAN International Conference on Funtional Programming*. Citeseer, 2010.

Curt J Sampson. Experience report: Haskell in the'real world': writing a commercial application in a lazy functional lanuage. *ACM Sigplan Notices*, 44(9):185–190, 2009.

Michael L Scott. Programming language pragmatics. 2009.

D. A. Turner. The sasl language manual. 1976.

Arie Van Deursen, Paul Klint ja Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* Väitöskirja, University of Oxford, 1971.

David A Watt. *Programming language design concepts*. John Wiley & Sons, 2004.

Lesley Wevers. Persistent functional languages: toward functional relational databases. *Proceedings of the 2014 SIGMOD PhD symposium*, sivut 21–25. ACM, 2014.