Aalto-yliopisto
Perustieteiden korkeakoulu
Informaatioverkostojen koulutusohjelma

Laiska evaluointi

Kandidaatintyö

13. joulukuuta 2017

Atte Keinänen

Aalto-yliopisto
Perustieteiden korkeakoulu
Informaatioverkostojen koulutusohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä:	Atte Keinänen
Työn nimi:	Laiska evaluointi
Päiväys:	13. joulukuuta 2017
Sivumäärä:	24
Pääaine:	Informaatioverkostot
Koodi:	SCI3026
Vastuuopettaja:	Apulaisprofessori Juho Kannala
Työn ohjaaja(t):	Vanhempi yliopistonlehtori Juha Sorva (Tietotekniikan laitos)

Laiska evaluointi on ohjelmointikielten suoritusta ohjaava periaate, jossa ajatuksena on turhan ja liian aikaisen lausekkeiden evaluoinnin välttäminen. Selvitin työssäni laiskan evaluoinnin yhteydessä käytettävää käsitteistöä, laiskan evaluoinnin historiaa ja sen nykysovellutuksia. Lisäksi selvitin, millaisia etuja ja heikkouksia laiskaa evaluointiin liittyy.

Työn menetelmänä on narratiivinen kirjallisuuskatsaus. Aineistona käytin akateemisia artikkeleita, oppikirjoja, blogikirjoituksiaja ohjelmointikielten dokumentaatioita. Etuja ja heikkouksia selvitin myös haastattelujen avulla.

Käsitteistöä selvittäessäni päädyin siihen, että laiska evaluointi on yläkäsite usealle eri evaluointisemantiikalle. Evaluointisemantiikat määrittävät, milloin ja miten funktion parametrilausekkeet evaluoidaan. Historian osalta tuli selville, että laiskan evaluoinnin kehitys nivoutuu tiiviisti Haskell-ohjelmointikielen tutkimukseen, ja että Haskell on ainoa yleisessä käytössä oleva kieli, joka käyttää laiskaa evaluointia laajasti. Totesin, että nykysovellutuksia on useita erilaisia, ja niitä on käytössä kaikissa suosituissa ohjelmointikielissä ja niiden apukirjastoissa. Näitä sovellutuksia ovat muun muassa laiskat listat ja generaattorit. Etuja ja heikkouksia tarkastellessa tuli esille, että Haskellissa laiska evaluointi vaikeuttaa sen ajonaikaisen suorituksen seuraamista ja aiheuttaa ajoittaisia muistivuotoja. Toisaalta rajatummista sovellutuksista oli hyviä kokemuksia.

Laiska evaluointi on ollut suuressa roolissa etenkin funktionaalisen ohjelmointityylin kehittymisessä. Nykyään laiska evaluointi vaikuttaa väistyvän ohjelmointikielien suunnittelua määräävänä periaatteena, mutta sitä osataan käyttää hyödyksi sovellutuksissa, joissa laajemman käytön aiheuttamia ongelmia ei tule.

Avainsanat:	laiska evaluointi, normaalijärjestyksessä evaluointi, call-by-value, call-by-need, evaluointisemantiikka, evaluointistrategia, evaluointijärjestys, funktionaalinen ohjelmointi, Haskell
Kieli:	Suomi

Sisältö

1 Johdanto

Laiska evaluointi on 1970-luvulta juontuva ohjelmointikielten suoritusta ohjaava periaate. Sen taustalla on ajatus siitä, että ohjelmakoodia suorittaessa kannattaisi välttää turhaa tai liian aikaista lausekkeiden evaluointia. Vastaavasti evaluointi kannattaa tehdä vasta, jos lausekkeen arvoa tarvitaan. Näin vältytään sekä tekemästä turhaa laskentaa että evaluoimasta sellaisia lausekkeita, jotka voisivat tuottaa ajonaikaisia ongelmia.

Periaatetta on luontevaa havainnollistaa esimerkillä. Kuvitellaan, että haluat laatia ohjelman jollain laiskan evaluoinnin periaatteita noudattavalla ohjelmointikielellä. Tämä ohjelma käsittelee listaa, joka koostuu aritmeettisista laskutoimituksista, ja tulostaa listan alkioita käyttäjälle. Ohjelmakoodi voisi näyttää seuraavalta:

Listaus 1 Pseudokielinen esimerkki listaa käsittelevästä ohjelmasta

```
1 list = [2*3, 5^10, 100/2]
2 print(list[0])
3 print(list[2])
```

Laiskan evaluoinnin periaatteita noudattaen ohjelman suoritus voisi tapahtua seuraavasti:

- Kun lista luodaan komennolla `list = [2*5, 5^10, 100/2]`, mitään laskutoimitusta ei vielä evaluoida, koska se olisi liian aikaista.
- Kun listan alkioita tulostetaan, niiden arvot (25 ja 50) lasketaan vasta juuri ennen tulostamista.
- Koska lausekkeen `5^10` arvoa ei tulosteta ohjelman suorituksen aikana, sitä ei koskaan evaluoida. Näin ylimääräiseltä työltä säästytään.

Laiska evaluointi on periaatteena monille ohjelmoijille vieras, koska suosituista yleiskäyttöisistä ohjelmointikielistä ainoastaan Haskell käyttää sitä suoritusmallinaan. Merkittävästi yleisempi on ahneeksi evaluoinniksi kutsuttu periaate, jossa ohjelmakoodin suoritus etenee ohjelmakoodin kuvaamassa järjestyksessä, eikä laskentaa viivästetä myöhempään ajanhetkeen.

Monissa moderneissa ohjelmointikielissä on kuitenkin mahdollista käyttää laiskaa evaluointia joissain kielen ominaisuuksissa. Näistä esimerkkejä ovat Clojuren laiskat listarakenteet, Scalan laiskat instanssimuuttujat ja C++:n laiskat futuurit. Lisäksi joissain ohjelmointikielissä on suosittuja apukirjastoja, joilla laiskaa evaluointi voi simuloida. Tällainen on esimerkiksi JavaScript-yhteisössä suosittu Immutable.js, joka tarjoaa laiskasti evaluoituja listarakenteita.

Ohjelmointiyhteisön sisällä ymmärrys laiskasta evaluoinnista on vajavaista. Sen käytön hyödyistä ja haitoista on sirpaleista ja ristiriitaista tietoa. Lisäksi aihepiiristä keskusteleminen on hankalaa, koska laiskan evaluoinnin määritelmä ei ole yksiselitteinen. Näihin ongelmiin työni pyrkii vastaamaan seuraavien tutkimuskysymysten avulla:

1. Millaista käsitteistöä laiskan evaluoinnin yhteydessä käytetään?
2. Mitkä ovat laiskan evaluoinnin sovellutuksia historiallisesti ja nykypäivänä?
3. Millaisia etuja ja heikkouksia laiskaan evaluointiin liittyy?

Seuraavaksi käyn läpi tutkimuksessa käytetyt tutkimusmenetelmät joka tutkimuskysymyksen osalta. Sen jälkeen vastaan tutkimuskysymykseen 1 määrittelemällä keskeisimmät käsitteet luvussa 3. Tutkimuskysymykseen 2 vastausta pohjustan luvussa 4, joka käsittelee laiskan evaluoinnin historiaa. Se antaa lukijalle valmiudet ymmärtää laiskan evaluoinnin nykyisiä sovellutuksia, jotka ovat luvun 5 aiheena. Luvussa 6 vastaan tutkimuskysymykseen 3. Luvussa 7 vedän työtä yhteen sekä käsittelen tulevaisuudennäkymiä ja mahdollisia jatkotutkimuksen aiheita.

2 Tutkimusmenetelmät

Tutkimuskysymykseen 1 vastaan hakemalla yleisimpiä käsitteitä, joita laiskaa evaluointia käsittelevässä kirjallisuudessa esiintyy, hakemalla näille yleisimmät määritelmät, ja vertailemalla käsitteiden vaihtoehtoisia määritelmiä. Jäsentelen myös käsitteiden suhteita toisiinsa.

Tutkimuskysymyksessä 2 laiskan evaluoinnin merkityksen tutkiminen tarkoittaa, että tarkastelen (a) laiskan evaluoinnin varhaista historiaa pohjustuksenomaisesti, (b) laiskan evaluoinnin leviämistä moderneihin ohjelmointikieliin, (c) laiskan evaluoinnin kautta kehittyneitä sovellutuksia ohjelmointikielissä ja (d) laiskaa evaluointia käyttävien teknologioiden hyödyntämistä tutkimuksessa ja työelämässä.

Tutkimuskysymyksessä 3 selvitän sitä, millaisia subjektiivisia mielipiteitä laiskaa evaluointia hyödyntäviä teknologioita käyttävillä ihmisillä on sekä laiskasta evaluoinnista yleisesti, että kysymyksen 2 tarkastelussa esiin tulleista ohjelmointikielistä ja apukirjastoista.

Menetelmänä kaikkiin kysymyksiin vastaamisessa on narratiivinen kirjallisuuskatsaus. Tavoitteena on kartoittaa, millaista tietoa aiheesta tällä hetkellä on, ja tuoda sitä yhteen helppotajuisen narratiivin muotoon. Tällä tavoin luotu katsaus soveltuu hyvin materiaaliksi opetustyössä (Baumeister ja Leary, 1997, s. 312). Tämä sopii työni tavoitteisiin, sillä

tavoitteena on nostaa tietämystä laiskasta evaluoinnista ja helpottaa siitä keskustelemista.

Kysymyksiin 1 ja 2 vastatakseni materiaalina käytän ensisijaisesti akateemisia artikkeleita, joita aiheesta on kirjoitettu paljon. Hain artikkeleita ensisijaisesti Scopus-tietokannasta, jossa käytin seuraavaa hakulauseketta:

Listaus 2 Hakulauseke Scopus-tietokannasta

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

Hakulausekkeessa käytetyt hakusanat esitellään luvussa 3. Hakulauseke tuotti otsikosta, tiivistelmästä ja avainsanoista haettaessa 326 tulosta. Tämä oli sopiva määrä tuloksia tutkimuskysymysten kannalta relevanteimpien artikkeleiden valikointia ajatellen.

Huomattava osa tuloksista liittyi laiskan evaluoinnin matemaattisiin todistuksiin, jotka ovat olennaisia ohjelmointikielten kääntäjien suunnittelun kannalta, mutta eivät tämän työn rajauksen kannalta. Kuitenkin etenkin Haskell-ohjelmointikielen parissa työskentelevien ihmisten kirjoittamat artikkelit, joita tuloksissa oli myös paljon, osoittautuivat hyödyllisiksi.

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 blogikirjoituksilla. Niitä hyödynnän sellaisten uusien kehityskulkujen esittelyssä, joiden esiintyminen akateemisissa artikkeleissa on hyvin vähäistä. Käsitteiden määrittelyssä käytin myös alan yliopistotasoisia oppikirjoja lähteinä.

Kysymykseen 3 vastatakseni käytän yhdistelmää eri tiedonhakutapoja. Hyödynnän (a) kirjallisuuskatsauksen tiedonhaussa vastaan tulleita mielipidelatautuneita artikkeleita, (b) blogikirjoituksia ja (c) mielipiteitä, joita olen kysynyt työkavereiltani Slack-kommunikaatioalustan avulla. Blogikirjoituksissa painotan kirjoittajia, joilla on myös akateemista näyttöä aihepiiristä. Mielipidettä kysyessä pyydän vastaajia kertomaan, millaisia vahvuuksia tai heikkouksia he ovat kohdanneet, jos he ovat käyttäneet laiskaa evaluointia ohjelmointikielissä tai niiden apukirjastossa.

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

3 Käsitteiden määrittely

Käyn tässä osiossa läpi käsitteistöä, joka on olennaista laiskan evaluoinnin aihepiirin ymmärtämiselle. Ensiksi käsittelen funktionaalista ohjelmointia, ohjelmointityyliä, johon laiska evaluointi usein liitetään. Sen jälkeen käyn läpi evaluointisemantiikkoja, jotka määrittelevät ohjelmointikielten funktioiden ja funktiokutsujen noudattamia sääntöjä.

3.1 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on tapa rakentaa tietokoneohjelmia, joka on saanut inspiraationsa lambdakalkyylistä, matemaattisen logiikan mallista, jonka Alonzo Church laati 1930-luvulla (Church, 1932).

Scott (2009) sanoo, että tiukasti määriteltynä funktionaalisella tyylillä rakennettu ohjelma määrittelee ohjelman ulostulot niiden sisääntulojen funktiona. Funktiot ovat funktioita matemaattisessa mielessä, eli ne eivät muuta funktion ulkopuolista tilaa. Funktiot voivat kutsua toisiaan, ja siten ohjelmissa useat pienemmät funktiot yhdessä muodostavat ohjelman pääfunktion, joka muuntaa sisääntulot ulostuloiksi.

Seuraavat käsitteet liittyvät läheisesti funktionaaliseen ohjelmointiin:

- *Deklaratiivinen ohjelmointi* on funktionaalisen ohjelmoinnin yläkategoria, jossa keskiössä on sen kuvaaminen, mitä tietokoneen halutaan tekevän. Deklaratiivisille ohjelmointikielille on ominaista korkea abstraktiotaso ja se, että ohjelmoijan on luontevaa muotoilla ongelmansa kielen tarjoamilla abstraktioilla. Deklaratiivisen ohjelmoinnin vastakohta on imperatiivinen ohjelmointi, jossa puolestaan keskitytään siihen, miten tietokoneen kuuluisi suorittaa halutut tehtävät.

- *Deterministisyys* tarkoittaa sitä, että tietyillä sisääntuloilla ohjelman ulostulo on aina sama riippumatta ajanhetkestä tai muista tekijöistä.

- *Referentiaalinen läpinäkyvyys* (eng. *referential transparency*) tarkoittaa sitä, että ohjelmointikielen lausekkeen korvaaminen sen arvolla ei muuta ohjelman ulostuloa (Hudak, 1989). Referentiaalisesti läpinäkyvät lausekkeet ovat myös deterministisiä.

- *Sivuvaikutus* (eng. *side effect*) tarkoittaa sitä, että ohjelmointikielen yhden aliohjelman kutsuminen joko vuorovaikuttaa ohjelman ulkopuolisen maailman kanssa tai vaikuttaa ohjelman muiden aliohjelmien palauttamiin arvoihin ohjelman myöhemmässä suoritusvaiheessa. Jos ohjelmassa on sivuvaikutuksia, se ei ole deterministinen.

- *Puhdas funktio* (eng. *pure function*) tarkoittaa funktiota, joka on sivuvaikutukseton, viittauksiltaan läpinäkyvä ja deterministinen.

- *Puhdas funktionaalinen ohjelmointi* (eng. *pure functional programming*) tarkoittaa ohjelmointityyliä, jossa käytetään ainoastaan puhtaita funktioita.

Käsitteistä deterministisyys, referentiaalinen läpinäkyvyys ja sivaikutuksettomuus ovat merkitykseltään osin päällekkäisiä, ja niitä kaikkia käytetään yleisesti funktionaalisen ohjelmoinnin luonteen kuvaamiseen.

Funktionaalinen ohjelmointityyli voi yksinkertaisimmillaan tarkoittaa sivuvaikutusten välttämistä, ja monet ohjelmointikielet tarjoavat tätä helpottavia työkaluja. Tällaisia kieliä ovat esimerkiksi Scala, Clojure ja Lisp. Jotkin kielet, kuten Haskell ja Miranda, rakentuvat pitkälti puhtaan funktionaalisen ohjelmointityylin ympärille.

3.2 Evaluointisemantiikat

Evaluointisemantiikka määrittelee säännöt ohjelmointikielen funktiokutsujen eri vaiheiden evaluoinnille. Käytännössä se vastaa yhteen tai useampaan seuraavista kysymyksistä:

- Milloin funktiolle funktiokutsussa annetut parametrilausekkeet evaluoidaan?
- Millä tavoin evaluoidut parametrilausekkeiden arvot välitetään funktion ohjelmakoodille?

Kuvassa 1 on eriteltynä työni kannalta merkittävimmät evaluointisemantiikat. Kuvasta myös näkyy kuinka evaluointisemantiikat voi hahmottaa puuhierarkiana, jossa kukin hierarkian taso vastaa johonkin kysymykseen evaluoinnin noudattamista säännöistä.

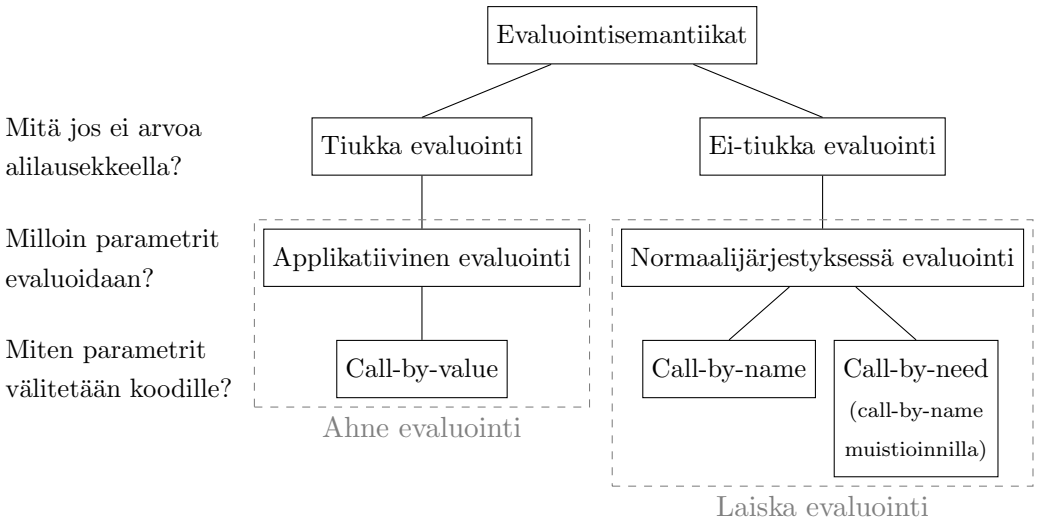
Seuraavissa alaluvuissa kuvaan kunkin evaluointisemantiikan tyypillisimmän määritelmän ja kerron, mitä muita merkityksiä kyseiselle käsitteelle välillä kirjallisuudessa liitetään.

3.2.1 Tiukka ja ei-tiukka semantiikka

Tiukka ja ei-tiukka semantiikka ovat etenkin Haskell-yhteisön käyttämiä termejä, jotka eivät suoranaisesti ota kantaa funktiokutsun eri vaiheiden evaluointiin. Ne ovat kuitenkin merkitykseltään läheisiä varsinaisille evaluointisemantiikoille, joten ne on luontevaa esitellä muiden evaluointisemantiikkojen yhteydessä.

HaskellWikin (2017) mukaan *tiukka semantiikka* (eng. *strict semantics*) tarkoittaa sitä, että ohjelmointikielen lausekkeella ei voi olla arvoa, jos millä tahansa sen alilausekkeella ei ole arvoa. *Ei-tiukassa semantiikassa* (eng. *non-strict semantics*) puolestaan lausekkeilla voi olla arvo, vaikka alilausekkeilla, joista nämä lausekkeet koostuvat, ei olisi arvoa.

Lauseke ei saa arvoa, jos sen evaluointi aiheuttaa esimerkiksi ikuisen silmukan tai ohjelman suorituksen lopettavan virheilmoituksen.



Kuva 1: **Evaluointisemantiikkojen keskinäisiä suhteita puuhierarkialla havainnollistettuna.** Tiukka ja ei-tiukka semantiikka eivät ole varsinaisia evaluointisemantiikkoja, mutta ne on sisällytetty luvussa 3.2.1 kuvatuista syistä johtuen. Ahne ja laiska evaluointi ovat epätarkasti määriteltyjä yleiskäsitteitä, joten niiden alle kuuluu useampi evaluointisemantiikka.

3.2.2 Applikatiivinen evaluointi ja normaalijärjestyksessä evaluointi

Scottin (2009) mukaan *applikatiivinen evaluonti* (eng. *applicative evaluation*) tarkoittaa funktion parametrilausekkeiden evaluointia ennen funktion ohjelmakoodin suorituksen aloittamista. Listauksessa 3 on applikatiivisesta evaluoinnista Python-ohjelmointikielellä luotu esimerkki.

Listaus 3 Esimerkki applikatiivisesta evaluoinnista Pythonilla

```
1  # Funktio, jonka suoritus jatkuu ikuisesti ja joka ei koskaan palauta arvoa
2  def noreturn(x):
3    while True:
4      x = -x
5    return x
6
7  # Luodaan lista, jonka ainoan alkion muodostava lauseke ei koskaan palauta arvoa
8  # Lauseke evaluoidaan välittömästi, joten ohjelman suoritus ei etene
9  list = [noreturn(1)]
10 print "Tämä ei koskaan tulostu"
```

Normaalijärjestyksessä evaluointi (eng. *normal-order evaluation*) puolestaan tarkoittaa, että parametrilausekkeet evaluoidaan vasta sitten, kun niitä tarvitaan. Evaluoimattomat parametrilausekkeet välitetään ohjelmakoodille, ja vasta sitten kun ohjelmointikoodissa käytetään parametrien arvoa, lausekkeet evaluoidaan. Listaus 4 demonstroi normaalijär-

jestyksessä evaluointia Haskell-ohjelmointikielessä.

Listaus 4 Esimerkki normaalijärjestyksessä evaluoinnista Haskellilla

```haskell
-- Funktio, jonka suoritus jatkuu ikuisesti ja joka ei koskaan palauta arvoa
noreturn :: Integer -> Integer
noreturn x = negate (noreturn x)

-- Luodaan lista, jonka ainoan alkion muodostava lauseke ei koskaan palauta arvoa
-- Lauseketta ei vielä evaluoida listan määrittelyhetkellä
list = [noreturn 1]

-- Haskell-ohjelman suoritus tapahtuu `main` -päälausekkeessa
main = do
  -- Listan pituutta laskettaessa alkion arvoa ei tarvita,
  -- joten alkion muodostavaa lauseketta ei tarvitse evaluoida
  length list >>= print
  "Tämä tulostuu" >>= print
  -- Listan ensimmäisen arvon tulostaminen aiheuttaa lausekkeen evaluoinnin,
  -- joten ohjelman suoritus ei enää etene
  head list >>= print
  "Tämä ei koskaan tulostu" >>= print
```

Applikatiivinen evaluointi noudattaa tiukkaa semantiikkaa, sillä kaikilla funktioparametreilla (eli alilausekkeilla) on oltava arvo ennen kuin funktio itsessään evaluoidaan. Siten applikatiivisen evaluoinnin voi ajatella olevan tiukan semantiikan alakategoria evaluointisemantiikkojen hierarkiassa.

Vastaavasti normaalijärjestyksessä evaluointi, jossa evaluointia viivästetään, täyttää eitiukan semantiikan kriteeristön, koska parametri voi jäädä funktiokutsussa evaluoimatta, jos sitä ei tarvita funktion ohjelmakoodissa. Siinä tilanteessa ei ole merkitystä, olisiko tällä parametrilla ollut arvoa vai ei.

3.2.3 Parametrimoodit

(Scott, 2009) kutsuu parametrien ohjelmakoodille välittämistä kuvaavia periaatteita *parametrimoodeiksi* (eng. *parameter mode*). Niistä laiskan evaluoinnin kannalta relevantteja ovat seuraavat:

- *Call-by-value*, jossa funktiota kutsuttaessa parametrilauseke evaluoidaan ennen funktion ohjelmakoodia, ja arvo on käytettävissä parametria vastaavassa muuttujassa funktion ohjelmakoodissa. Jos parametrilausekkeena on ollut muuttuja, muut-

tujan arvo kopioidaan uuteen muuttujaan funktion ohjelmakoodia varten. (Scott, 2009)

- *Call-by-name*, jossa parametrilausekkeet sijoitetaan suoraan funktion ohjelmakoodiin niihin kohtiin, joissa argumentteihin käytetään. Parametrilauseke evaluoidaan uudestaan joka kerta, kun ohjelmakoodissa tarvitaan kyseisen argumentin arvoa. (Ariola et al., 1995)

- *Call-by-need*, jossa parametrilauseke evaluoidaan vasta, kun funktion ohjelmakoodi tarvitsee sen arvoa ensimmäistä kertaa. Kun parametrilauseke on evaluoitu, se pidetään muistissa sitä varten, jos ohjelmakoodi tarvitsee argumenttia uudestaan. Tämä on yksi *muistioinnin* (eng. *memoization*) muoto. (Ariola et al., 1995)

Call-by-value noudattaa applikatiivista evaluointia siinä missä call-by-name ja call-by-need evaluoidaan normaalijärjestyksessä. Siten parametrimoodit voi nähdä applikatiivisen evaluoinnin ja normaalijärjestyksessä evaluoinnin alakategorioina.

3.2.4 Laiska ja ahne evaluointi

Laiska evaluointi (eng. *lazy evaluation*) esiintyy kirjallisuudessa tarkoittaen samaa kuin joko normaalijärjestyksessä evaluointi tai jokin sen alakategorioista. Myös Scott (2009) toteaa, että laiskaa evaluointia käytetään usein eräänlaisena yleiskäsitteenä useammille toisiaan muistuttaville evaluointisemantiikoille.

Ahne evaluointi (eng. *eager evaluation*) määritellään usein laiskan evaluoinnin vastakohdaksi. Se voi kontekstista riippuen tarkoittaa samaa kuin esimerkiksi applikatiivinen evaluointi tai call-by-value -parametrimoodi.

Laiska ja ahne evaluointi ovat mielestäni molemmat käsitteinä ongelmallisia, koska ne ovat alttiita sekaannuksille sen suhteen, mitä niillä milloinkin tarkoitetaan. Siksi pyrin tulevissa luvuissa käyttämään tarkemmin määriteltyjä evalointisemantiikkojen käsitteitä silloin kun se on mahdollista.

4 Laiskan evaluoinnin kehitys

Normaalijärjestyksessä evaluoinnin periaate syntyi 1970-luvulla. Sarja julkaisuja loi pohjaa ajatukselle “laiskoista” funktionaalisista kielistä työkaluna käytännönläheiseen ohjelmistokehitykseen. Ajatus esiteltiin ensimmäisenä matemaattisesti lambdakalkyylin, funktionaalisen ohjelmoinnin kannalta keskeisen matemaattisen teorian, näkökulmasta (Wadsworth, 1971). Viisi vuotta myöhemmin julkaistiin toisistaan riippumatta kolme ar-

tikkelia (Henderson ja Morris Jr, 1976; Friedman, 1976; Turner, 1976), joissa esiteltiin normaalijärjestyksessä laiskaa evaluointia funktionaalisen ohjelmoinnin perspektiivistä.

4.1 Haskellin ja puhtaan funktionaalisen ohjelmoinnin kehitys

Hudak et al. (2007) kuvaa, kuinka 1980-luvun puolivälissä laiskaa evaluointia (joka tuolloin käytännössä tarkoitti käytännössä normaalijärjestyksessä evaluointia) hyödyntävien ohjelmointikielien määrä kasvoi nopeasti. Useimmat kielistä soveltuivat vain kapeaan määrään käyttökohteita, ja niiden käyttäjämäärä oli pieni. 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 piirissä olevien evaluointisemantiikkojen tieteellistä kehitystä. Alkuvaiheen kehitystyö oli onnistunutta, ja Haskellista kehittyi etenkin tietojenkäsittelytieteen akateemisen tutkimuksen parissa suosittu kieli.

Haskell ei määritelmällisesti ole normaalijärjestyksessä evaluoitu ohjelmointikieli, vaan jo Haskell 1.1 -raportissa (Hudak et al., 1991) se määriteltiin ainoastaan ei-tiukkaa semantiikan määritelmän täyttäväksi kieleksi. Käytännössäkään kielessä funktioiden parametreja ei aina evaluoida vasta kun ohjelmakoodi niitä tarvitsee, vaan Haskellin kääntäjät tekevät useita nopeusoptimointeja suorittamalla tiettyjä osia koodista tiukan semantiikan säännöllä.

Useimmat kielen rakenteet noudattavat kuitenkin oletusarvoisesti call-by-need –evaluointisemantiikan sääntöjä, eli funktioiden argumenttilausekkeet evaluoidaan vasta sitten niitä kuin tarvitaan, ja evaluoidut arvot muistioidaan. Listauksessa 5 on esimerkki siitä, miten call-by-need -semantiikka näkyy käytännössä Haskellissa.

Listaus 5 Esimerkki call-by-need -semantiikasta Haskellissa

```
> -- Tämä esimerkki ajetaan Haskellin komentoriviversiossa eli REPLissä
> -- Luodaan kokonaisluvuista koostuva lista, jolla on kaksi alkiota
> let list = [5 * 5, 5 + 5] :: [Integer]
> -- Tulostetaan listan arvo ja huomataan, että alkioita ei vielä ole evaluoitu
> -- Merkki `_` kertoo, että arvoa ei ole evaluoitu
> :sprint list
[_,_]
> -- Tulostetaan listan ensimmäinen arvo
> -- (erillistä `print` -kutsua ei REPlissä tarvita)
> head list
25
> -- Nyt ensimmäinen arvo on muistioitu, eli se voidaan palauttaa suoraan muistista,
> -- ja vain toinen arvo on evaluoimatta
> :sprint list
[25,_]
> -- Tulostetaan listan kaikki arvot
> -- Ensimmäinen arvo palautetaan muistista ja toinen evaluoidaan
> list
[25,10]
> :sprint list
[25,10]
```

Call-by-need -evaluointisemantiikka aiheuttaa sen, että ohjelmointikielen lausekkeiden evaluointi etenee erilaisessa järjestyksessä kuin applikatiivisesti evaluoiduissa kielissä. Call-by-need -semantiikan kielissä järjestys, jossa ohjelmakoodin rivit on kirjoitettu, ei vielä kerro suoritusjärjestystä, vaan suoritusjärjestys määräytyy dynaamisesti ohjelman suorituksen aikana.

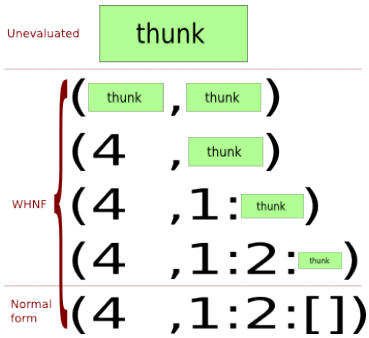
Yksi merkittävä käytännön seuraus tästä on, että IO-operaatioita, eli esimerkiksi näytölle tulostamista tai käyttäjän syötteen odottamista, ei pystytä suorittamaan halutussa järjestyksessä pelkkien tavallisten funktiokutsujen avulla. Siksi Haskelliin on kehitetty uniikkeja ohjelmointikielen rakenteita, kuten *monadit*, joilla IO-operaatioita ja niiden järjestystä pystyy kontrolloimaan erillään muusta ohjelmakoodista.

Toinen seuraus call-by-need -evaluointisemantiikasta on se, että funktion parametrien arvoja ei ole mahdollista jälkikäteen muuttaa, koska suoritusjärjestys määräytyy dynaamisesti ja parametrin arvon muutokset voisivat tapahtua arvaamattomassa järjestyksessä. Siten funktionaalisen ohjelmoinnin ajatusmaailma etenkin referentiaalisen läpinäkyvyyden osalta sopii hyvin call-by-need -semantiikan kanssa yhteen. Haskell tukee puhtaasti funktionaalista ohjelmointityyliä kattavasti, sillä Haskellin funktioissa ei ole sivuvaikutuksia muutoin kuin monadien tarjoaman rajapinnan kautta.

4.2 Haskellissa kääntäjän suunnittelussa tehdyt tekniset valinnat

Laiskan evaluoinnin kokonaisvaltaisessa tarkastelussa on olennaista myös käydä läpi sitä, miten se on ohjelmointikielen kääntäjissä toteutettu, ja millaisia seurauksia sillä on ohjelman suorituksen aikaiseen käyttäytymiseen. Haskell-ohjelmointikielen ylivoimaisesti suosituin kääntäjä on ollut Glasgow Haskell Compiler (lyhyemmin GHC), joka on ollut 1990-luvun alusta lähtien ollut olennainen osa ohjelmointikielen kehityskaarta.

Kuten edellä mainittiin, Haskell-yhteisö tarkoittaa laiskalla evaluoinnilla yleensä call-by-need -evaluointisemantiikkaa. Call-by-need -semantiikka on GHC:ssä toteutettu siten, että ohjelman suorituksen aikana kielen lausekkeista muodostetaan dynaamisesti verkkoa, jossa yksittäistä evaluoimatonta lauseketta kutsutaan *tyngiksi* (eng. *thunk*) (Hudak et al., 2007). Näitä tynkiä evaluoidaan sitä mukaan, kun jokin ohjelman IO-operaatio tarvitsee niiden arvoja. Kuvassa 2 näytetään, miten erään yksinkertaisen tietorakenteen evaluointi etenee.



Kuva 2: **Lausekkeen (4, [1, 2]) evaluointi Haskellissa.** Ensiksi lauseke on kokonaan evaluoimatta. Se evaluoidaan asteittain, kunnes tynkiä ei ole jäljellä. Kokonaan evaluoidusta lausekkeesta sanotaan, että se on *normaalimuodossa*. Lauseketta, joka on ainakin osittain evaluoitu, kutsutaan *heikoksi päänormaalimuodoksi* (eng. *Weak Head Normal Form*). Kuvan lähde on WikiCommons (2007).

Tämä toteutus eroaa monista applikatiivisesti evaluoiduista kielistä etenkin siinä, että tynkien muodostava verkko voi itsessään viedä ohjelman suoritusaikana huomattavasti muistia. Lisäksi se, että jokin IO-operaatio tarvitsee jonkin tyngän arvoa, voi aiheuttaa hetkellisen hidastumisen ohjelman suoritukseen, jos sisäkkäisiä evaluoimattomia tynkiä on kertynyt paljon. Kumpaakaan näistä ilmiöistä ei esiinny applikatiivisen evaluoinnin kielissä, sillä niissä lausekkeet suoritetaan siinä järjestyksessä missä ne on ohjelmakoodiin kirjoitettu, ja tynkiä ei tarvita.

4.3 Leviäminen muihin ohjelmointikieliin

Haskell on nykypäivänä suosituista ohjelmointikielistä ainoita, joissa applikatiivinen evaluointisemantiikka on rakennettu perustavanlaatuiseksi osaksi kielen toteutusta. Kuitenkin monet Haskellissa esitellyt ideat ovat levinneet myös muihin kieliin. Käsittelen näitä ideoita ja niiden sovelluksia eri ohjelmointikielissä seuraavassa luvussa.

5 Sovellutuksia ohjelmointikielissä ja apukirjastoissa

Laiskaa evaluointia voi käyttää myös yksittäisissä kielen ominaisuuksissa. Tämä osio käy läpi, millaisia sovellutuksia laiskalle evaluoinnille on suosituissa ohjelmointikielissä. Luku on jäsennelty sovellutuksittain, jotta ohjelmointikielien väliset toteutuserot tulevat selkeästi ilmi. Sovellutukset on koottu taulukkoon 1, ja alaluvuissa niitä käydään läpi tarkemmin.

Sovellutus	**Evaluointisemantiikka**	**Toteutukset**	
		Ohjelmointikielissä	**Apukirjastoissa**
Laiskat sekvenssit	Applikatiivinen evaluointijärjestys	Clojure, Haskell, Python, Scala	Immutable.js
Call-by-name -muuttujat	Call-by-name	Scala (funktion parametrina)	-
Call-by-need -muuttujat	Call-by-need	Scala (luokan instanssimuuttujana)	-
Laiskat futuurit/lupaukset	Call-by-need	C++1	Fluture (JavaScript-kirjasto)
Käyttäjän laatimat kontrollirakenteet	Applikatiivinen evaluointi tai kääntämisvaiheen makrot	Clojure (makrot), Haskell, Scala (makrot, kokeellinen tuki)	-
Generaattorit	Funktion suorituksen keskeyttäminen (vrt. applikatiivinen evaluointi)	JavaScript, Python	-

Taulukko 1: **Sovellutukset koottuna.** Toteutus ohjelmointikielessä tarkoittaa, että kieli tukee sovellutusta joko syntaksinsa tai standardikirjastonsa kautta. Toteutus apukirjastossa tarkoittaa, että sovellutusta emuloidaan ohjelmointikielen tarjoamilla välineillä.

5.1 Läpikäydyt ohjelmointikielet

Valitsin tarkasteluun viisi ohjelmointikieltä, jotka ovat keskenään erilaisia ja jotka ovat suosittuja sekä akateemisessa ympäristössä että liike-elämässä. Haskellin lisäksi valitsin

seuraavat neljä kieltä:

- *Clojure* on Lisp-kieltä muistuttava yleiskäyttöinen, funktionaalista ohjelmointityyliä tukeva ohjelmointikieli. Clojure-lähdekoodi kääntyy Java-tavukoodiksi ja on täysin yhteensopiva Java-kirjastojen kanssa.
- *JavaScript* on ainoa verkkoselainten yleisesti tukema ohjelmointikieli, ja lisäksi JavaScriptiä käytetään runsaasti myös palvelinohjelmointiin. Se tukee funktionaalista ohjelmointityyliä rajoitetusti.
- *Python* on sekä imperatiivista että rajatusti funktionaalista ohjelmointia tukeva yleiskäyttöinen ohjelmointikieli. Se on suosittu niin tieteellisessä käytössä kuin verkkokehityksessä.
- *Scala* on sekä funktionaalista että imperatiivista ohjelmointia tukeva yleiskäyttöinen ohjelmointikieli. Scala-lähdekoodi kääntyy Clojuren tapaan Java-tavukoodiksi, ja kieli on täysin yhteensopiva Java-kirjastojen kanssa.

5.2 Laiskat sekvenssit

Laiska sekvenssi on listatietorakenne, joissa listan arvoja evaluoidaan sitä mukaan kun ohjelmakoodi tarvitsee niitä.

Clojure 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.

JavaScript-ohjelmointikielessä ei itsessään ole tukea laiskoille sekvensseille. Kuitenkin Facebookin julkaisema suosittu *Immutable.js*-apukirjasto 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.

Myös Python tukee laiskoja sekvenssejä *generaattorilausekkeiden* (eng. *generator expression*) avulla ja Scalassa *virtojen* (eng. *stream*) avulla. Molempien kielien toteutukset ovat

listarakenteisiin erikoistuneita versioita generaattoreista, joita käsitellään tarkemmin alaluvussa 5.7.

5.3 Call-by-need -muuttujat

Scala tukee *call-by-need muuttujia* luokkien instanssimuuttujina. Niitä käytetään lisäämällä `lazy` -avainsana instanssimuuttujan määrittelyn alkuun. Kyseisen muuttujan arvon määrittävä lauseke evaluoidaan vasta sitten, kun muuttujaa käytetään ensimmäistä kertaa. Arvo muistioidaan, eli jos muuttujaa tarvitaan uudelleen, arvoa ei enää evaluoida uudelleen.

5.4 Call-by-name -muuttujat

Scala tukee *call-by-name -muuttujia* funktion parametreina. Ne 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.

5.5 Laiskat futuurit

Futuuri (eng. *future*, *promise* tai *deferred*) tarkoittaa ohjelmointikielen rakennetta sellaisten operaatioiden kuvaamiseen, jotka voivat olla evaluoimatta, joiden evaluointi on kesken tai joiden evaluointi on jo palauttanut arvon. Futuureja käytetään operaatioiden keskinäiseen koordinoitiin ja synkronointiin.

Useimmissa ohjelmointikielissä futuurit ovat ahneita, eli niiden kuvaamaan operaation evaluointi aloitetaan futuurin luontihetkellä. *Laiskat futuurit* ovat futuureja, joissa vasta se, kun ohjelmakoodi haluaa käyttää futuurin kuvaaman operaation arvoa, aiheuttaa operaation evaluoinnin.

C++ tukee laiskoja futuureja, ja niitä voi luoda kutsumalla `std::async` -funktiota `std::launch:deferred` -asetuksella. Funktiokutsu palauttaa `std::future` -olion, jonka `get` -metodi aiheuttaa operaation suorituksen.

5.6 Käyttäjän laatimat kontrollirakenteet

Kontrollirakenteella tarkoitetaan kielen natiivisyntaksin, kuten `if`, `for` tai `while`, kaltaisten rakenteiden luomisen itse. 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.

Clojure puolestaan tukee *makroja*, joiden avulla kielen kääntäjää pystyy laajentamaan ohjelmakoodista käsin, ja sen myötä kieleen voi luoda omia kontrollirakenteita.

5.7 Generaattorit

Generaattorit ovat ohjelmointikielen rakenne, jolla funktio pystyy keskeyttämään suorituksensa ja palauttamaan arvon useasti. Tätä käytetään etenkin silmukkarakenteiden kanssa, jossa jokainen silmukan ajo tuottaa jonkin uuden arvon. Generaattorit muistuttavat läheisesti laiskaa evaluointia, koska niiden avulla pystyy esimerkiksi pyytämään listatietorakenteiden seuraavia alkioita alkio alkiolta tarpeen mukaan.

JavaScript ja Python toteuttavat generaattorit siten, että ne palauttavat *iteraattorin*, jolta voi pyytää seuraavaa arvoa. Myös kielten listarakenteet noudattavat iteraattorirajapintaa, joten sen myötä listoja ja generaattoreilla luotuja iteraattoreita voi käsitellä monilla samoilla komennoilla.

6 Subjektiiviset edut ja haitat

6.1 Ongelmanratkaisun helpottuminen tai vaikeutuminen

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.

HaskellWiki (2006) toteaa, että laiskuus mahdollistaa kielen käyttäjälle sen, että hän voi operoida päättymättömien ja määrittelemättömien tietorakenteiden kanssa, niin kauan kun hän ei yritä käyttää määrittelemättömiä osia. Näin ohjelmoija pystyy käsittelemään esimerkiksi matemaattisia todistuksia sujuvammin.

Karvonen (2016) puolestaan väittää, että laiskuus ei merkittävästi muuta sitä, millaisia ongelmia kielellä pystyy kuvaamaan. Useat sellaiset kirjastot, joiden väitetään olevan toteutettavissa vain laiskasti evaluoidun kielen avulla, ovat hänen mukaansa helposti toteutettavissa ei-laiskalla kielellä simuloimalla 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 rekursiivisesti määritellyille tietotyypeille, joka löytyy esimerkiksi kilpailevasta ML-ohjelmointikielestä. Tämän myötä kielellä ei ole mahdollista esimerkiksi luoda tietotyyppiä luonnollisten lukujen kuvaamiseen tai tyyppiä luonnollisten lukujen muodostamalle listalle.

Jones (2010) sanoo, että seuraava versio Haskellista voisi oletusarvoisesti noudattaa tiukkaa semantiikkaa, ja kielen käyttäjä voisi ottaa laiskan evaluoinnin aina tarpeen mukaan käyttöön. Hänestä tapa, jolla kieltä käytetään, ei merkittävästi muuttuisi tämän muutoksen myötä. Hänen mukaansa laiskaa evaluointia tärkeämpää on se, että kieli jatkossakin rakentuu puhtaasti funktionaalisen ohjelmoinnin ympärille.

6.2 Suorituskyky

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 muis-

tinkäytön ennustettavuus on heikko.

6.3 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. Kohtaa, jossa arvoa käytetään, voi kuitenkin kirjoittajien mielestä olla vaikeaa löytää. Tämä liittyy hänen mukaansa myös ajonaikaisten virheiden paikantamisen vaikeuteen.

Scott (2009) sanoo, että yleisesti ottaen applikatiivinen evaluointi on lähtökohtaisesti normaalijärjstyksessä evaluointia parempi valinta, sillä ohjelman suoritusjärjestys on tuolloin selkeä ja eksplisiittisesti ilmaistu.

6.4 Koettu arvo

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

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.

Mohan (2010) sanoo, että hän ylläpitää laajaa Haskell-ohjelmaa työssään, ja sen kokemuksen pohjalta Haskell on muita kieliä parempi tietynlaisiin ohjelmiin. Hän arvostaisi,

jos Haskell olisi “oletuksena tiukasti evaluoitu”. Hän myös toteaa viitaten Peytonin (2010) esitykseen että normaalijärjestyksessä evaluoinnin ansiosta Haskellissa on monadien kaltaisia ilmaisuvoimaisia abstraktioita, jotka muutoin olisivat ehkä jääneet keksimättä.

7 Yhteenveto ja johtopäätökset

Aloitan käymällä läpi, millaisia vastauksia sain tutkimuskysymyksiini.

Käsitteitä määritellessä selvisi, että laiskaan evaluointiin liittyen käytetään paljon päällekkäistä käsitteistöä, mikä hankaloittaa aiheeseen tutustumista. Tässä työssä päädyin käyttämään paljon normaalijärjestyksessä evaluoinnin ja sen vastakohdan applikatiivisen evaluoinnin käsitteitä, koska ne ovat yksiselitteisesti määriteltyjä. Laiska evaluointi ja ahne evaluointi osoittautuivat yleiskäsitteiksi, joilla voidaan kuvata useita eri evaluointistrategioita.

Tietojenkäsittelytieteessä normaalijärjestyksessä evaluointi on selkeästi ollut keskeistä etenkin funktionaalisen ohjelmoinnin kehityksen kannalta. Se on kannustanut rakentamaan kieliä, jossa muuttujien arvot pysyvät vakiona ja funktiot käyttäytyvät ennakoitavaksi, koska tällöin komentojen suoritusjärjestystä voi muuttaa ja suoritusta voi viivästää.

Normaalijärjestyksessä evaluointia osataan käyttää yhä kypsemmin ja kypsemmin: sen ja applikatiivisen evaluoinnin välillä osataan tasapainotella, ja laiskaa evaluointia käytetään rajatummin kuin aikaisemmin. Tämä liittyy siihen, että laiskan evaluoinnin laajamittainen käyttö aiheuttaa ongelmia sekä nopeuden että muistinkäytön ennustettavuuden suhteen. Rajatuissa sovellutuksissa näitä ongelmia ei pääse syntymään.

Vaikuttaa myös siltä, että normaalijärjestyksessä evaluointi itsessään ei merkittävästi helpota ohjelmointiongelmien kuvaamista ja ratkaisemista. Oikeastaan laajamittainen käyttö vaikeuttaa muistivuotojen identifioimista sekä suorituksen seuraamista. Nykyiset työkalut esimerkiksi Haskellin suorituksen seuraamiseen ovat hyvin puutteelliset verrattuna moniin applikatiivisesti evaluoituihin kieliin.

7.1 Metodologisia puutteita

Mielipidetutkimuksen otos oli pieni ja tutkimusmenetelmä hieman epämääräinen. Oli kätevää kysyä kokemuksia työkavereilta firman viestintätyökalun käyttää, mutta keskustelu ei ollut luonteeltaan kovin jäsentynyttä. Kirjallisuudessakin mielipiteitä ja kokemuksia oli hajanaisesti eri aiheista. Oli vaikeaa luoda yksittäiseen ala-aiheeseen, kuten laiskan evaluoinnin nopeushyötyihin ja -haittoihin, monipuolista näkemystä.

7.2 Alan ongelmia

Toivoisin, että etenkin Haskell-yhteisö kävisi vastaisuudessa enemmän keskustelua siitä, millaista käsitteistöä he käyttävät evaluointisemantiikkoihin liittyen. Haskell-wiki, joka on monelle Haskellin käyttäjälle tärkeä oppimateriaali, ei käytä lainkaan alan vakiintuneita call-by-value ja call-by-need -käsitteitä, vaikka ne ovat kuvaavia ja selkeästi määriteltyjä termejä. Sen sijaan Haskell-yhteisö puhuu paljon ei-tiukasta semantiikasta, joka ei ota ohjelmakoodin evaluoinnin etenemiseen suoraan kantaa. Minulle tuli Haskell-wikiä lukiessa usein sellainen olo, että asioita ei selitetä siinä riittävän selkeästi.

Haskell-wikin toinen ongelma on, että siinä laiskan evaluoinnin aiheuttamia ongelmia ei käsitellä riittävästi. Päinvastoin laiskaa evaluointia ja ei-tiukkaa semantiikkaa ylistetään varsin kritiikittömästi. Tämä johtaa monia Haskelliin tutustuvia harhaan, ja voi johtaa siihen, että Haskellin käytön aloittaneet kohtaavat isoja ongelmia kieltä käyttäessään.

7.3 Tulevaisuudennäkymiä

Yhtenä uutena kiinnostavana trendinä on totaalisesti funktionaalinen ohjelmointi, jossa funktiot ovat täydellisesti määriteltyjä, eli ne saavat poikkeuksetta arvon (Turner, 2004). Päättymättömiä silmukoita tai loputtomia rekursioita ei siis näissä kielissä esiinny.

Näiden kielien kannalta sillä, onko kieli applikatiivisesti vai normaalijärjestyksessä evaluoituva, ei ole merkitystä. Idris-niminen kieli, joka on viimeisenä muutamana vuotena saavuttanut suosiota, on valinnut applikatiivisen evaluoinnin. Kielen luojien perusteena valinnassa on, että applikatiivisesti evaluoidut ohjelmat ovat ennakoitavampia suorituskyvyltään (Idris Github, 2017).

7.4 Mahdollisia tutkimuskysymyksiä

Olisi kiinnostavaa tutkia, miten työssä esiteltyjä laiskan evaluoinnin sovellutuksia voisi toteuttaa sellaisiin ohjelmointikieliin, jossa niitä ei vielä ole. Näitä kieliä voisivat olla esimerkiksi Java ja Go.

Olisi myös hyödyllistä saada selville, miten applikatiivista evaluointia käyttävissä kielissä nopeuteen ja muistinkäyttöön liittyvistä ongelmista voisi päästä eroon. Mikä olisi esimerkiksi paras tapa varoittaa käyttäjää, että hänen kirjoittamansa ohjelmakoodi saattaa aiheuttaa muistinkäytöllisiä ongelmia? Voisiko myös kehittää parempia ohjelmien suorituksen tarkkailemiseen tarkoitettuja työkaluja, jotka toisivat nämä ongelmat havainnollisesti ilmi?

7.5 Oma oppimiseni

Koin kandityön tekemisen itselleni hyödylliseksi. Tutustuin evaluointisemantiikkoihin ja funktionaaliseen ohjelmointiin paljon syvemmällä tasolla kuin aikaisemmin. Pystyn nyt tekemään valistuneempia valintoja siitä, millainen ohjelmointikieli soveltuu parhaiten käsillä olevaan ongelmaan, ja kykenen hyödyntämään laiskaa evaluointia rajatuiden ongelmien ratkaisemisessa.

Oli silmiä avaavaa huomata, kuinka epäselvää tietojenkäsittelytieteiden käsitteistö usein on. Koen, että jatkossa minun on tärkeää kiinnittää enemmän huomiota siihen, että kommunikoin yksiselitteisiä käsitteitä käyttäen työssäni, jotta saan vältettyä epäselvyyksiä.

Lähteet

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen ja Philip Wadler. A call-by-need lambda calculus. *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, sivut 233–246, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199507. URL `http://doi.acm.org/10.1145/199448.199507`.

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

Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, sivut 346–366, 1932.

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

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. Performance/laziness, 2006. URL `https://wiki.haskell.org/Performance/Laziness`. Viitattu 10.12.2017.

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.

Idris Github. Idris Unofficial FAQ, 2017. URL https://github.com/idris-lang/Idris-dev/wiki/Unofficial-FAQ#why-isnt-idris-lazy. Viitattu 10.12.2017.

Simon Peyton Jones. Wearing the hair shirt: A retrospective on Haskell, 2010. URL http://www.cs.nott.ac.uk/~gmh/appsem-slides/peytonjones.ppt. Viitattu 10.12.2017.

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

Ravi Mohan. Keskusteluketju artikkelille On Hiring Haskell People, Hacker News, 2010. URL https://news.ycombinator.com/item?id=1924061. Viitattu 10.12.2017.

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.

David A Turner. Total Functional Programming. *J. UCS*, 10(7):751–768, 2004.

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.

WikiCommons. Kuva Thunk-layers.png, 2007. URL https://commons.wikimedia.org/wiki/File:Thunk-layers.png. Viitattu 12.12.2017.