Aalto-yliopisto
Perustieteiden korkeakoulu
Informaatioverkostojen koulutusohjelma

Laiska evaluointi funktionaalisessa ohjelmoinnissa

Kandidaatintyö, keskeneräinen luonnos

11. helmikuuta 2016

Atte Keinänen

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 kasitellään tarkemmin luvussa 2. Kirjallisuudessa käsitteet kuitenkin esiintyvät myös toistensa synonyymeina.

tuotti otsikosta, tiviistelmästä 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.

Pitää vielä selvittää, mitkä tutkimusmenetelmät kuvaisivat parhaiten sitä mielipideselvitystä, joka tehdään tutkimuskysymykseen 2 vastaamiseksi.

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

1.4 Työn rakenne

Kirjoitan myöhemmin auki, kun sisällysluettelo hahmottuu:

Esittelen työn rakennetta perustellen sen menetelmien ja tutkimuskysymysten avulla. Kerron, että käsittelen ensin katsausmaisesti laiskan evaluoinnin varhaista historiaa, minkä jälkeen poraudun ensimmäiseen tutkimuskysymykseen laiskasta evaluoinnista nykypäivänä. Sitten, kun lukijalla on hyvä yleiskuva nykypäivän sovellutuksista, on sujuvaa siirtyä puhumaan yksittäisten sovellutusten yksityiskohdista sekä vahvuuksista ja heikkouksista.

2 Laiskan evaluoinnin perustaa

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 ohjelmointikieliä. 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 ohjelmointikiel-

ten 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 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 lausekkellakaan 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" -periaattella. 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, vai viittaanko samaan esimerkkiin myöhemmin uudelleen.

Taulukko 1: Esimerkki kielen semantiikan vaikututuksesta evaluointijärjestykseen

Graafireduktio on tapa ei-laiskan evaluoinnin 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 evaluoinnin “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.

Lähteet

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

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

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.

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.

Michael L Scott. Programming language pragmatics. 2009.

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

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.