Laajennoksia tyypitettyyn lambdakalkyyliin

Kaisa Krannila

Helsinki 13. 2. 2003
Esitelmä seminaarissa Tyyppiteoria ja ohjelmointikielet
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Esitän tässä esitelmässä joitakin laajennoksia puhtaaseen yksinkertaisesti tyypitettyyn lamdbakalkyyliin. Esitykseni pyrkii olemaan tiivistelmä Benjamin C. Piercen teoksen Types and Programming Languages ([Pie02]) luvuista 11 ja 13, ja oletan lukijalle tutuksi aikaisempien lukujen sisällön, mm. em. kalkyylin tarkemman kuvauksen.

Esittämäni lisäykset voi jakaa periaatteeltaan kolmeen luokkaan, ja käsittelen kutakin luokkaa omassa luvussaan. Tarkastelen ensin joitakin ns. johdettuja muotoja, yksinkertaisia syntaktisia muunnoksia, joissa kussakin esitellään jotakin kompleksista syntaktista risuaitaa korvaamaan jotakin helpommin kirjoitettavaa ja semanttisesti ekvivalenttia.

Toiseksi laajennamme kieltä lisäämällä siihen joukon tyyppejä. Periaatteessa tällöin pitäisi todistaa, että lisäystenkin jälkeen kielen tyyppijärjestelmä on turvallinen, ts. että tyypittyvä termi on joko arvo tai evaluoitavissa (etenemisteoreema), ja että tyypittyvän termin evaluointitulos on aina tyypittyvä (säilymisteoreema). Käytännössä kuitenkin sivuutan todistukset.

Kolmanneksi laajennamme kieltä tavalla, joka aiheuttaa muutoksia siinä, miten määrittyy kielellemme operationaalisen semantiikan antavan tilakoneen tila. Aikaisemmin tilaksi on riittänyt ohjelmalaskurin arvo; nyt siihen on lisättävä myös muistin kulloinenkin sisältö. Myös etenemis- ja säilymisteoreema muuttuvat; niiden periaatteellinen merkitys kuitenkin pysyy samana, ja niiden pätevyys olisi todistettava nytkin tehtäville lisäyksille.

2 Johdettuja muotoja

([Pie02, 11.1 - 11.5])Lisätään kieleen aluksi muutama tyyppi: Teoreettisissa tarkasteluissa saattaa toisinaan olla kätevää abstrahoida tyypistä kaikki yksilöivät ominaisuudet pois. Tätä varten voidaan kielen syntaksin tyyppivalikoimaan liittää tyyppi tai tyyppejä, jolle ei määritellä mitään operaatioita, ja jonka mahdollisia alkioita ei voi suoraan nimetä. Tämmöistäkin tyyppiä voidaan silti hyödyntää aivan mielekkäissä ilmauksissa; esim. jos A on tulkitsematonta tyyppiä esittävä metamuuttuja, λx:A.x on A:n identiteettifunktio, tyypiltään A $\rightarrow$ A.

Tulkituista tyypeistä yksinkertaisin mahdollinen on tyyppi Unit: on tasan yksi elementti, vakio unit, joka on tätä tyyppiä. Operaatioita tyypille ei ole määritelty. Sellaisena se muistuttaa esim. Javan void-tyyppiä. Unit-tyyppiä

voidaan hyödyntää *peräkkäisyyden* formalisointiin: Liitämme kielen syntaksiin termien joukkoon termin t_1; t_2. Epämuodollisesti sanoen termi tulkitaan siten, että evaluoidaan ensin termi t_1 ja sitten termi t_2. Arvo, johon evaluointi päättyy, on sama kuin jos olisi evaluoitu vain t_2. Miksi ohjelmoija sitten vaivautuisi kirjoittamaan ohjelmaansa moisen termin t_1, ja miksi se vielä evaluoitaisiinkin, jollei se vaikuta arvoon, joksi termi evaluoituu? Tyypillisesti evaluoinnin *sivuvaikutusten* takia; näihin palaamme luvussa .

Peräkkäisyyden tulkinta voidaan formalisoida kahdellakin Unit-tyyppiä hyödyntävällä tavalla: Kieleen voidaan lisätä evaluointi- ja tyypityssäännöt

$$\frac{t_1 \;\rightarrow\; t_1'}{t_1 \,;\, t_2 \;\rightarrow\; t_1' \,;\, t_2} \qquad \text{E-SEQ}$$

$$\text{unit}; \, t_2 \;\rightarrow\; t_2 \qquad \text{E-SEQNEXT}$$

$$\frac{\Gamma \vdash t_1 \,:\, \text{Unit} \qquad \Gamma \vdash t_2 \,:\, T_2}{\Gamma \vdash t_1 \,;\, t_2 \,:\, T_2} \qquad \text{T-SEQ}$$

Toinen, yksinkertaisempi mahdollisuus on olla lisäämättä mitään uusia sääntöjä ja sen sijaan tulkita t_1; t_2 lyhenteeksi termistä $(\lambda x{:}\text{Unit}.t_2).t_1$, jossa x ei ole mikään t_2:n vapaa muuttuja. Call by value -strategian mukaan ensin evaluoidaan parametri t_1, sitten sijoitetaan tulos x:n paikalle t_2:ssa (eli ei minnekään, koska x ei ole t_2:n vapaa muuttuja), sitten evaluoidaan t_2 sijoituksen jälkeisessä muodossaan (eli entisellään). Eli yhteensä evaluoinnin tulos on sama kuin olisi evaluoitu pelkkä t_2.

Nämä kaksi tulkintaa ovat semanttisesti ekvivalentit. Peräkkäisyys on esimerkki *johdetusta muodosta*: sen tyypitys- ja evaluointisäännöt on mahdollista palauttaa kielen perustavampiin operaatioihin, abstraktioon ja sovellukseen. Johdettu muoto on vain kielen pintasyntaksin laajennos: otetaan käyttöön ohjelmoijalle helpompi ja luontevampi tapa ilmaista jotakin, joka kyllä jo kielen olemassaolevallakin syntaksilla on ilmaistavissa. Koska kielen semantiikka tai tyyppijärjestelmä ei rikastu mitenkään, ei ole tarvetta todistaa esim. tyyppiturvallisuuden kaltaista teoreemaa johdetulle muodolle. -Toinen yksinkertainen johdettu muoto on *villin kortin* käyttö abstraktiossa: jos abstraktion rungossa ei esiinny parametrin tyyppistä muuttujaa, ei ole tarpeen antaa parametrille mitään spesifiä nimeä abstraktiossa: voimme kirjoittaa $\lambda\_{:}S.t$, sen sijaan että kirjoittaisimme $\lambda x{:}S.t$ tai $\lambda y{:}S.t$, tms.

Toisinaan on kätevää tai tarpeen voida "julistaa" (ascribe) termi tietyntyyppiseksi. Mahdollistamme tämän kielessä lisäämällä termivalikoimaan termin t as T. Evaluointisäännöt ovat yksinkertaiset: termiä t evaluoidaan, kunnes jäljellä on v as T, jolloin lopuke "as T" vain heitetään pois. Tyypityssääntö puolestaan on

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{T-ASCRIBE}$$

Entä jos ohjelmoija valehtelee termin tyypin? Jos $\Gamma \vdash t_1 : T_1$, ja ohjelmoija kirjoittaakin syntaktisesti ihan mahdollisen termin t_1 as T_2? Evaluointi toimii väitetystä tyypistä riippumatta, ei siis jumitu, mutta tyypityssääntöjen mukaan termi t_1 as T_2 ei ole tyypittyvä. Otaksuakseni tässä on esimerkki staattisen tyypityksen konservatiivisuudesta: tyypintarkistin ilmoittaa tyyppivirheestä, joka ei kuitenkaan ajon aikana saisi aikaan mitään katastrofaalista.

Voidaan osoittaa, että tyypinjulistuksessakin on kysymys johdetusta muodosta. Sitä voidaan hyödyntää esimerkiksi dokumentoimaan ohjelmaa. Mielenkiintoisempi käyttötarve syntyy, kun kieleen lisätään alityypitys, eli termillä voi olla useita tyyppejä: tyypintarkistinta kehotetaan ottamaan huomioon näistä tyypeistä vain jokin. Samantapaiseen tarkoitetun tyypin rajaavaan käyttöön tutustumme vaihtuvarakenteisen tietuetyypin yhteydessä.

Termi voidaan sitoa nimeen let-sidonnalla let $x{=}t_1$ in t_2. Call by value - strategiasta seuraa, että termi $t_1$on evaluoitava täysin, ennen kuin itse let:n evaluointi suoritetaan; seuraavat evaluointisäännöt ilmentävät tätä:

$$\text{let } x{=}t_1 \text{ in } t_2 \;\rightarrow\; [x \mapsto v_1]t_2 \qquad \text{E-LETV}$$

$$\frac{t_1 \;\rightarrow\; t_1'}{\text{let } x{=}t_1 \text{ in } t_2 \;\rightarrow\; \text{let } x{=}t_1' \text{ in } t_2} \qquad \text{E-LET}$$

(tyypityssäännöt:)

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x{=}t_1 \text{ in } t_2 : T_2} \qquad \text{T-LET}$$

Onko let-sidonta johdettu muoto? Osin on, osin ei. Abstraktion ja sovelluksen yhdistelmällä saadaan kyllä sanottua sama kuin let-sidonnalla:

$$\text{let } x{=}t_1 \text{ in } t_2 \quad =^{def} \quad (\lambda x{:}T_1.t_2)\ t_1$$

Jos rakennettaisiin kääntäjä, joka suoraviivaisesti suorittaisi let-termien konversion sovelluksiksi ylläolevan määritelmän mukaisesti, pitäisi kuitenkin ratkaista vielä yksi ongelma: miten jäsennin tietäisi, että muuttujan x tyypiksi täytyy merkitä T_1? Ei jäsennin sitä tiedäkään; tieto saadaan vasta tyypintarkastusvaiheessa, kun lasketaan termin t_1 tyyppi. Jos let tulkitaan johdetuksi muodoksi, se ei kuitenkaan ole samalla tapaa yksinkertainen puhtaasti syntaktinen muunnos kuin aiemmin käsitellyt johdetut muodot.

3 Rakenteiset tietotyypit

([Pie02, 11.6 - 11.10])Määritellään *tietue* järjestetyksi kokoelmaksi nimettyjä termejä: $\{l_i{=}t_i^{\,i \in 1..n}\}$. Tietueen nimiöitä merkitään siis $l_i^{\,i \in 1..n}$, ja tietueen tyyppi on $\{l_i : T_i^{\,i \in 1..n}\}$. Termien (joita tunnetusti kutsutaan yleensä kentiksi) tyyppejä ei siis ole mitenkään rajoitettu. *Projektio* t.l on operaatio, jota käytetään palauttamaan tietueen kentän l arvo. Tietue itse on täysin evaluoitu (siis arvo), kun sen jokainen kenttä on.

Määritelmässämme kenttien järjestyksellä on väliä; ts. tietueet, joissa on samannimiset kentät eri järjestyksessä, ovat erityyppisiä. (Tässä suhteessahan eri ohjelmointikielet poikkeavat toisistaan.)

Tietueesta voidaan johtaa erikoistapauksena *monikot*, sallimalla nimiöiksi luonnolliset luvut, ja jättämällä nimiö vain kirjoittamatta näkyviin sen ollessa luonnollinen luku. Tietue puolestaan voidaan nähdä *vaihtelevan tietuetyypin* (variant) erikoistapauksena, jossa tietueilla on vain yksi vaihtoehtoinen rakennemalli. Vaihtelevaa tietuetyyppiä käytetään heterogeenisten tietokokoelmien alkion tyyppinä: kaikissa alkioissa ei tarvitse olla samoja kenttiä. Syntaktisesti tyyppi rakennevaihtoehtoineen ilmaistaan $<l_i = T_i^{\,i \in 1..n}>$ eli jokaiselle rakennevaihtoehdolle annetaan nimiö [1].

Sen, että termi t on tulkittava vaihtelevan tietuetyypin T ilmentymävaihtoehdoksi, jonka nimiö on l, ilmaisemme julistamalla termin tietuetyypin ilmentymäksi (tai *"liputtamalla"*, tagging), eli termillä <l=t> as T.

Liputusrakenteen tyypinjulistus (as T) on periaatteellisesti olennainen lisäys: Jos ohjelmassa on useita vaihtelevia tietuetyyppejä, on mahdollista, että

[1] Pierce puhuu tässä mielestäni virheellisesti kentän nimiöistä (field label); kuitenkin asiayhteyden perusteella vaikuttaa ilmeiseltä, että kyseessä ovat rakennevaihtoehtojen nimiöt.

niitten rakennevaihtoehtoina esiintyy samoja tyyppejä. Esimerkiksi jos T1 = <lankapuhno:T3, kotiosoite:T4> ja T2 = <lankapuhno:T3, kanny:T5>, ei pelkkä <lankapuhno=t> riittäisi kertomaan termistä t, onko se tyyppiä T1 vai T2 . Koska haluamme - ainakin vielä - pitää kiinni tyypin yksikäsitteisyydestä, ratkaisemme ongelman käyttämällä aiemmin esitettyä tyypinjulistus-rakennetta. Myöhemmin, alityypityksen käsittelyn yhteydessä, ehkä näemme toisenlaisen ratkaisun samaan ongelmaan; toistaiseksi kuitenkin liputamme, jolloin tietueen evaluointi- ja tyypityssäännöt ovat seuraavat:

$$\frac{t_i \;\rightarrow\; t_i'}{< l_i \;=\; t_i > \;\text{as T} \;\rightarrow < l_i \;=\; t_i' > \;\text{as T}} \qquad \text{E-VARIANT}$$

$$\frac{\Gamma \vdash t_j \;:\; T_j}{\Gamma \vdash < l_j \;=\; t_j > \;\text{as} \;< l_i \;:\; T_i^{\,i\in 1..n} > \;:\; < l_i \;:\; T_i^{\,i\in 1..n} >} \qquad \text{T-VARIANT}$$

(Yllä ilmeisesti yhteinen alaindeksi j sitoo premissin T_j:n ja johtopäätöksen l_j:n toisiinsa siten, että l_j on rakennevaihtoehdon T_j nimiö.)

Vaihtelevarakenteisia tietueitten käsittely esim. jossakin funktiossa tyypillisesti riippuu siitä, minkärakenteinen tietue kulloinkin on kysymyksessä. Haarautuminen eri käsittelyvaihtoehtoihin rakennetyypin mukaan koodataan case-rakenteella: case t of $<l_i{=}x_i> \Rightarrow t_i^{\,i\in 1..n}$. Casessa siis luetellaan kaikki mahdolliset rakennevaihtoehdot ja vastaavat toimintatavat (=nuolta seuraavat termit). Ennen tietueen rakennetta vastaavan toiminnon suorittamista tietue evaluoidaan, sitten evaluoinnin tuloksena saatu arvo sijoitetaan muuttujan x_i paikalle termissä t_i. Eli formalisoituna, tyypityksen kera:

$$\begin{array}{c}\text{case } (\; < l_j = v_j > \;\text{ as T}) \text{ of } \;< l_i = x_i > \Rightarrow t_i^{\,i\in 1..n} \\ \rightarrow\; [x_j \mapsto v_j]t_j\end{array} \qquad \text{E-CASEVARIANT}$$

$$\frac{t_0 \;\rightarrow\; t_0'}{\begin{array}{c}\text{case } t_0 \text{ of } \;< l_i \;=\; x_i > \Rightarrow t_i^{\,i\in 1..n} \\ \rightarrow\; \text{case } t_0' \text{ of } \;< l_i \;=\; x_i > \Rightarrow t_i^{\,i\in 1..n}\end{array}} \qquad \text{E-CASE}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 \;:< l_i : T_i^{\,i\in 1..n} > \\ \text{for each } i \qquad \Gamma,\, x_i : T_i \vdash t_i \;:\; T\end{array}}{\Gamma \vdash \;\; \text{case } t_0 \text{ of } \;< l_i \;=\; x_i > \Rightarrow t_i^{\,i\in 1..n} \;:\; T} \qquad \text{T-CASE}$$

(Pikku harjoituksena voisimme miettiä, miksei case-rakenteessa liputeta termiä t tyypillään, ts. miksei oikean kulmasulun perässä lue “as T”.)

4 Rekursio yksinkertaisesti tyypitetyssä kalkyylissä

([Pie02, 11.11])Voidaan osoittaa ([Pie02], luku 12), ettei yksinkertaisia tyyppejä käyttäen (s.o. joka termillä enintään yksi tyyppi) voida tyypittää mitään lauseketta, joka voi johtaa päättymättömään laskentaan. Rekursiota ei näin ollen voi toteuttaa fix-funktiolla, kuten saattoi tehäd tyypittämättömässä kalkyylissa. Sen sijaan lisäämme kieleen primitiivin fix, joka itsessään ei ole termi, vaan muodostaa sellaisen vasta toisen termin kanssa. fix on muodostin, jota voidaan käyttää muodostamaan mm. rekursiivisia funktioita. Koska en ole saanut tolkkua sen evaluointisäännöstä E-FIXBETA sen kummemmin kuin Piercen aihetta koskevasta suorasanaisesta esityksestäkään, en valitettavasti voi kertoa aiheesta sen ymmärrettävämmin kuin luetella säännöt:

Evaluointi:

$$\frac{t_1 \;\rightarrow\; t_1'}{\text{fix } t_1 \;\rightarrow\; \text{fix } t_1'} \qquad \text{E-FIX}$$

$$\text{fix } (\lambda x{:}T_1.t_2) \rightarrow [\, x \;\mapsto\; (\text{fix } (\lambda x{:}T_1.t_2))\,]\, t_2 \qquad \text{E-FIXBETA}$$

Tyypitys:

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1} \qquad \text{T-FIX}$$

5 Laskennan sivuvaikutuksista: viittausmuuttujat

([Pie02, luku 13]) Käytännön ohjelmoinnissa ei useinkaan ole tavoitteena niin- kään selvittää laskennan lopputulos vaan saada aikaan *sivuvaikutuksia*, esimerkiksi viittausmuuttujan arvon päivittyminen, erilaiset syöttö- ja tulostustoimet, kontrollin siirtyminen ohjelman toiseen osaan. Ohjelma voi koostua peräkäisistä evaluoinneista, joitten tulokset eivät vaikuta toisiinsa eivätkä muutenkaan ole järin kiinnostavia.

Esimerkkinä sivuvaikutusten lisäämisestä kalkyyliin tarkastelemme viittausmuuttujia. Muutetaan laskennan mallia siten, että otetaan käyttöön joukko

paikkoja (locations). *Varastot* (store) ovat osittaisfunktioita paikkojen joukolta arvojen joukolle. Näin saadaan abstraktilla tasolla mallinnettua muisti (tarkemmin sanottuna lähinnä keko). Tilasiirtymäkoneen tilaa ei riitä ilmaisemaan enää pelkkä ohjelmalaskurin arvo (termi), vaan tila on pari (ohjelmalaskuri, varasto), eli laskennan tilaan täytyy liittää kuvaus siitä, miten kulloinkin sijainteja on yhdistetty arvoihin. Kaikki evaluointisäännöt täytyy muuttaa vastaavasti: Säännön argumenttina on evaluoitavan termin lisäksi kulloinenkin varasto, ja sääntö palauttaa myös varaston. Eli, jos merkitään μ:llä varastoa, emme kirjoita enää t $\rightarrow$ t' vaan t$|\mu \rightarrow$ t'$|\mu$'. [2]

Jotta varastoja pääsisi muuttelemaan ja varastofunktion arvoja käyttämään, lisätään kieleen termit ref t (viittauksen muodostaminen), !t (paikan sisällön palauttaminen), t_1:=t_2 (viitatun paikan sisällön muuttaminen). Paikat kuuluvat kielen arvojen joukkoon ja sellaisina myös termeihin; säännöissä paikkoja esittää metamuuttuja l.

Viittaus muodostetaan siten, että evaluoidaan t:tä kunnes jäljellä on arvo, valitaan paikka l, joka ei kuulu tämänhetkisen varaston lähtöarvoihin, ja muodostetaan uusi varasto edellisen varaston ja parin (l, v) yhdisteestä:

$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{\text{ref } t_1 \mid \mu \rightarrow \text{ref } t_1' \mid \mu'} \qquad \text{E-REF}$$

$$\frac{l \notin dom(\mu)}{\text{ref } v_1 \mid \mu \rightarrow l \mid (\mu,\ l \mapsto v_1)} \qquad \text{E-REFV}$$

Sijoituksessa evaluoidaan t1 paikaksi, t2 arvoksi, ja yhdistetään arvo sijaintiin (eli varasto muuttuu toiseksi). Se, että t1 evaluoituu juuri paikaksi, hoidetaan tyypityssäännöillä (jotka tuonnempana):

$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{t_1 := t_2 \mid \mu \rightarrow t_1' := t_2 \mid \mu'} \qquad \text{E-ASSIGN1}$$

$$\frac{t_2 \mid \mu \rightarrow t_2' \mid \mu'}{v_1 := t_2 \mid \mu \rightarrow v_1 := t_2' \mid \mu'} \qquad \text{E-ASSIGN2}$$

[2] Luvusta 13.3 eteenpäin Pierce kutsuu varastoksi (store) vuoroin paikkojen joukkoa, vuoroin osittaisfunktiota paikkojen joukolta arvojen joukolle. Koetan pitäytya johdonmukaisesti jälkimmäisessä merkityksessä. Tällöin ei ole mahdollista puhua esim. "varaston muuttumisesta", kun otetaan käyttöön uusi paikka.

$$l := \mathrm{v}_2 \mid \mu \rightarrow \mathrm{unit} \mid [\, l \mapsto \mathrm{v}_2 \,]\, \mu \qquad \text{E-ASSIGN}$$

Arvon palautus !t on yksinkertaista: Evaluoidaan t arvoksi. Jos arvo on paikka, termi evaluoituu vastaavaksi varastofunktion arvoksi:

$$\frac{\mathrm{t}_1 \mid \mu \rightarrow \mathrm{t}_1' \mid \mu'}{!\mathrm{t}_1 \mid \mu \rightarrow !\mathrm{t}_1' \mid \mu'} \qquad \text{E-DEREF}$$

$$\frac{\mu(l) = \mathrm{v}}{!l \mid \mu \rightarrow \mathrm{v} \mid \mu'} \qquad \text{E-DEREFLOC}[3]$$

Tyypitys hoituu kätevimmin siten, että lisätään tyyppien joukkoon Ref T, s.o. viittaukset erityyppisiin arvoihin. Nämä tyypit ovat /textitdomμ:n paikkojen tyyppejä. Kun sääntöä E-REFV kerran on sovellettu, kyseisen paikan tyyppi ei enää muutu: siihen voidaan liittää toisia arvoja, mutta vain samantyyppisiä. Systeemimme ei toistaiseksi sisällä minkäänlaista kertaalleen käyttöön otetun paikan vapautusta. Olkoon Σ funktio paikoilta tyypeille. Tällöin paikkoja ja viittauksia koskevat tyypityssäännöt voidaan formuloida näin:

$$\frac{\Sigma(l) = \mathrm{T}_1}{\Gamma \mid \Sigma \vdash l : \mathrm{Ref}\ \mathrm{T}_1} \qquad \text{T-LOC}[4]$$

$$\frac{\Gamma \mid \Sigma \vdash \mathrm{t}_1 : \mathrm{T}_1}{\Gamma \mid \Sigma \vdash \mathrm{ref}\ \mathrm{t}_1 : \mathrm{Ref}\ \mathrm{T}_1} \qquad \text{T-REF}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathrm{t}_1 : \mathrm{Ref}\ \mathrm{T}_{11}}{\Gamma \mid \Sigma \vdash !\mathrm{t}_1 : \mathrm{T}_{11}} \qquad \text{T-DEREF}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathrm{t}_1 : \mathrm{Ref}\ \mathrm{T}_{11} \quad \Gamma \mid \Sigma \vdash \mathrm{t}_2 : \mathrm{T}_{11}}{\Gamma \mid \Sigma \vdash \mathrm{t}_1 := \mathrm{t}_2 : \mathrm{Unit}} \qquad \text{T-ASSIGN}$$

[3] Koska varasto on vain osittaisfunktio, eikö premissinä pitäisi olla myös $l \in dom(\mu)$?
[4] Mikä T_1 mahtaa olla, jos $l \notin dom(\mu)$?

Tekemistämme lisäyksistä seuraa, että säilymis- ja etenemisteoreemat täytyy muotoilla toisin kuin ennen. Säilymisteoreema siis tarkoitti, että tyypittyvän termin evaluointitulos on tyypittyvä; seuraavalla muotoilulla saadaan aikaan toivottu tulos:

TEOREEMA [SÄILYMINEN]:

Jos

$\Gamma \mid \Sigma \vdash t : T$

$\Gamma \mid \Sigma \vdash \mu$

$t \mid \mu \rightarrow t' \mid \mu'$

niin, jollekin $\Sigma' \supseteq \Sigma$,

$\Gamma \mid \Sigma' \vdash t' : T$

$\Gamma \mid \Sigma' \vdash \mu'$.

Etenemisteoreeman uusi muoto on

TEOREEMA [ETENEMINEN]:

> Olkoon t suljettu tyypittyvä termi. Tällöin joko t on arvo tai kaikille varastoille μ, joille pätee $\emptyset \mid \Sigma \vdash \mu$, on olemassa termi t' ja varasto μ' siten, että $t \mid \mu \rightarrow t' \mid \mu'$.

Olettaisin, että päteäkseen teoreemat edellyttävät lisäksi, että sääntöön E-REFV yllä lisätään jotenkin Σ:n "päivitys".

LÄHTEET

[Pie02] Pierce, Benjamin C. *Types and Programming Languages.* the MIT Press, Cambridge, Massachusetts, 2002.