Lambdakalkyylin perusteita

Marja Huovinen

Helsinki 30.1.2003

Tyyppiteoria ja ohjelmointikielet - seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Lambda-kalkyyli on Alonzo Churchin (kollegoineen) 1920- ja 30-luvuilla kehittämä formaali menetelmä laskennan kuvaamiseen. Siinä laskenta on palautettu funktioiden määrittelyyn ja soveltamiseen. Lambdakalkyylia voidaan pitää sekä yksinkertaisena ohjelmointikielenä että matemaattisena järjestelmänä, jolle voidaan laatia helposti todistuksia. Sen vuoksi sitä hyödynnetään erityisesti ohjelmointikieliin ja tyyppijärjestelmiin liittyvässä tutkimuksessa. Funktionaaliset ohjelmointikielet perustuvat pitkälti lambdakalkyyliin.

Tässä seminaaritekstissä esitellään lambdakalkyylin ja tyyppijärjestelmien perusteet pohjautuen varsin tarkasti B.C.Piercen oppikirjan *Types and Programming Languages* lukuihin 5, 9 ja 12 [Pie02]. Myös käytetyt esimerkit ovat kirjasta.

2 Peruskäsitteitä

([Pie02], luku 5.1)
Lambda-kalkyylin syntaksi koostuu kolmenlaisista termeistä: *muuttujista*, *abstraktioista* ja *sovelluksista*. Jos muuttuja x on termi, myös sen abstraktio termistä t_1, jota merkitään $\lambda x.t_1$, on termi. Samoin termin t_1 soveltaminen termiin t_2, jota merkitään t_1, t_2, on termi. Alla taulukossa 3 termit on kuvattu kielioppina Piercen käyttämällä notaatiolla.

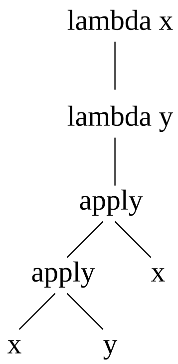
`t ::=`		termit:
	`x`	muuttuja
	λ`x.t`	abstraktio
	`t t`	sovellus

Taulukko 1:

Ohjelmointikielen *abstrakti syntaksi* voidaan kuvata puurakenteella. Todistusten ja esimerkkien hahmottamista helpottaa, jos ajattelee syntaksipuun lambdakalkyylin kieliopille. Esimerkkipuu on esitetty kuvassa 1.

Yllä esitettyä kielioppia voidaan pitää induktiivisesti määriteltynä joukkona abstraktin kieliopin puita. Olkoon $\mathcal{V}$ numeroituva joukko muuttujanimiä. Termien joukko on pienin joukko $\mathcal{T}$, jossa

1. x $\in \mathcal{T}$ kaikille x $\in \mathcal{V}$;
2. jos $t_1 \in \mathcal{T}$ ja x $\in \mathcal{V}$, niin $\lambda x.t_1 \in \mathcal{T}$;



Kuva 1: Esimerkki syntaksipuusta

3. jos $t_1 \in \mathcal{T}$ ja $t_2 \in \mathcal{T}$, niin $t_1\ t_2 \in \mathcal{T}$.

2.1 Notaatiosta

Kirjaimia `s`, `t` ja `u` käytetään ilmaisemaan metamuuttujia, jotka voidaan korvata millä tahansa termillä. Vastaavasti `x`, `y` ja `z` voivat tarkoittaa mitä tahansa muuttujaa.

Sulut oletetaan vasemmalle assosioituviksi eli `s t u` tarkoittaa samaa kuin `(s t) u`. Oletetaan myös, että abstraktiot *sitovat* muuttujia niin kauas oikealle kuin mahdollista. Esimerkiksi abstraktio λ`x. `λ`y. x y x` on sama kuin λ`x. (`λ`y. ((x y) x))`. Abstraktio λ`x.t` sitoo muuttujan `x` esiintymät abstraktion oikeanpuoleisessa osassa `t`.

3 Evaluaatio ja sijoittaminen

Merkitään evaluaatiorelaatiota "t_1 evaluoituu yhdessä askeleessa t_2:ksi" $t_1 \rightarrow t_2$. Esimerkiksi abstraktiossa

$$(\lambda x.t_{12})t_2 \rightarrow [x \rightarrow t_2]t_{12}$$

osa $[x \rightarrow t_2]t_{12}$ tarkoittaa termiä, joka saadaan kun kaikki `x`:n vapaat esiintymät t_{12}:ssa korvataan t_2:lla. Vapaita esiintymiä ovat kaikki `x`:n esiintymät, joissa abstraktio ei sido `x`:ää t_{12}:ssa. Yhden askeleen evaluaation tulee olla deterministinen.

Vapaat muuttujat voidaan määritellä termien määritelmän perusteella seuraavasti:

1. $FV(x) = \{x\}$;
2. $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$;

3. $FV(t_1 \quad t_2) = FV(t_1) \cup FV(t_2)$.

Termi, johon ei voida soveltaa mitään evaluaatiosääntöä, on *normaalimuodossa.* Normaalimuodossa olevien termien pitäisi olla vain arvoja. Kaikkien termien, jotka eivät ole normaalimuodossa, pitäisi olla johdettavissa normaalimuotoon evaluaatiolla. ([Pie02], luku 3) Termit, jotka eivät palaudu normaalimuotoon, *divergoivat.*

Sijoitusoperaatiot, joita evaluaatiossa tehdään, voidaan määritellä täsmällisesti. Määritelmässä tulee ottaa huomioon, että sidottujen muuttujien nimillä ei saa olla merkitystä. Lisäksi sidottujen ja vapaiden muuttujien nimet tulee pitää erillään, jotta vältetään *muuttujan vangitseminen* tyyliin $[x \rightarrow z](\lambda z.x) = \lambda z.z$. Termit, jotka eroavat toisistaan ainoastaan sidottujen muuttujien nimien osalta, voidaan vaihtaa keskenään missä tahansa kontekstissa. Määritelmä voidaan nyt muotoilla

$$
\begin{array}{lcl}
[x \rightarrow s]\; x & = & s \\
[x \rightarrow s]\; y & = & y \qquad \text{jos } y \neq x \\
[x \rightarrow s](\lambda y.t_1) & = & \lambda y.t_1 \qquad \text{jos } y = x \\
[x \rightarrow s](\lambda y.t_1) & = & \lambda y.[x \rightarrow s]t_1 \qquad \text{jos } y \neq x \text{ ja } y \notin FV(s) \\
[x \rightarrow s](t_1\; t_2) & = & [x \rightarrow s]t_1\; [x \rightarrow s]t_2
\end{array}
$$

4 Tyypittämätön lambdakalkyyli

Tyypittämättömässä *puhtaassa* lambdakalkyylissa ei ole valmiiksi sisäänrakennettuna vakioita tai operaattoreita kuten numerot, aritmeettiset operaatiot, ehtolauseet tai silmukat. Siinä ainoastaan sovelletaan funktioita toisille funktioille (myös argumentit ovat siis funktioita).

4.1 Operationaalinen semantiikka

([Pie02], luku 5.1)
Abstraktioita muotoa $(\lambda x.t_{12})t_2$, joiden oikeanpuolimmaisille osille suoritetaan sijoituksia, kutsutaan *redexeiksi.* Redexien evaluointia kutsutaan *beta-reduktioksi.* Evaluaatiostrategia määrittelee, mille termin redexille seuraava evaluaatioaskel voidaan tehdä. Evaluaatiostrategioita on useita erilaisia:

- *Täydessä beta-reduktiossa* mikä tahansa redex voidaan palauttaa koska tahansa.

- *Normal order* -strategiassa vasemmanpuoleinen uloin redex palautetaan ensimmäisenä.
- *Call by name* -strategia palauttaa vasemmalta alkaen kuten normal order, mutta abstraktioiden sisällä ei sallita palautuksia.
- *Call by value* -strategiassa vain uloimmat redexit palautetaan ja redex palautetaan vain, jos sen oikea puoli on jo palautettu arvoksi (eli oikeaan puoleen ei voida soveltaa evaluaatiosääntöjä).

Jatkossa oletetaan käytettäväksi call by value -strategiaa, joka on yleisimmin käytetty ohjelmointikielissä. Tämä vastaa siis käytäntöä, jossa funktiokutsun parametrit lasketaan valmiiksi ennenkuin funktiokutsua aletaan suorittaa.

Puhtaan tyypittämättömän lambdakalkyylin koko operationaalinen semantiikka voidaan tiivistää seuraavaan taulukkoon 3 ([Pie02] 5.3):

Syntaksi		
`t ::=`		Termit:
	`x`	muuttuja
	λ`x.t`	abstraktio
	`t t`	sovellus
`v ::=`		Arvot:
	λ`x.t`	abstraktioarvo
Evaluaatio		$t \to t'$
	$\dfrac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$	(E-APP1)
	$\dfrac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'}$	(E-APP2)
	$(\lambda \mathtt{x}.\ t_{12})\ v_2 \to [x \to v_2]t_{12}$	(E-APPABS)

Taulukko 2:

Koska call-by-value-strategia pysähtyy lambdaan, arvot ovat lambdatermejä. Evaluaatiossa on kahdenlaisia sääntöjä: *laskentasääntö* E-APPABS, joka tekee varsinaisen evaluoinnin ja *kongruenssisäännöt* E-APP1 ja E-APP2, jotka määrittelevät, missä kohtaa evaluointi tehdään. Nämä kolme sääntöä yhdessä määrittelevät evaluointijärjestyksen sovellukselle $t_1\ t_2$: ensin palautetaan kumpikin termi arvoksi kongruenssisäännöillä ja sitten sovelletaan arvoihin laskentasääntöä.

4.2 Ohjelmointi

Puhtaassa tyypittämättömässä lambdakalkyylissa funktiolla ei voi olla useampia argumentteja. Se voidaan kiertää käyttämällä korkeamman kertaluokan funktioita. Määritellään esimerkiksi Boolean-muuttujat seuraavasti:
`true` $= \lambda$`t.` λ`f. t;`
`false` $= \lambda$`t.` λ`f. f;`.
Nyt voidaan määritellä vertailuoperaatiota vastaava kombinaattorifunktio `test` $=$ λ`l.` λ`m.` λ`n. l m n`; joka palautuu `m`:ään kun `l` on `true` ja `n`:ään kun `l` on `false`.

Esimerkki: Churchin numeraalit

Määritellään *Churchin numeraalit* c_0, c_1, c_2 jne. seuraavasti:
$c_0 = \lambda$s. λz. z;
$c_1 = \lambda$s. λz. s z;
$c_2 = \lambda$s. λz. s (s z);
$c_3 = \lambda$s. λz. s (s (s z));
jne.

Joka numeroa siis vastaa c_n, joka ottaa argumentit `s` ja `z` ja soveltaa `s`:ää `z`:lle `n` kertaa. Esimerkiksi funktio `succ`, joka palauttaa numeron `n` seuraajan, voidaan nyt määritellä:

$$\text{succ} = \lambda\text{n. } \lambda\text{s. } \lambda\text{z. s (n s z);}$$

Tyypittämättömällekin lambdakalkyylille voidaan siis helposti ohjelmoida booleanit, numerot ja niihin liittyvät operaatiot.

Myös rekursio on mahdollinen, mutta siinä termi helposti divergoi. Kuitenkin esimerkiksi kiinteän pisteen kombinaattori `fix` (call-by-value-strategialle) on hyvin käyttökelpoinen:
fix $= \lambda$f. (λx. f (λy. x x y)) (λx. f (λy. x x y));

Termien kopioimisen avulla saadaan aikaan silmukoita vastaava toistokäyttäytyminen.

Oletetaan, että olemme määritelleet funktiot `comp` yhtäsuuruusvertailulle, `pred` numeraalin edeltäjälle ja `times` numeraalien kertolaskulle. Nyt kertoman laskemiselle tyypittämättömässä lambdakalkyylissa voidaan muodostaa rekursiivinen funktio:
`g` $= \lambda$`fct.` λ`n. test (comp n` c_0`)` $(\lambda x.c_1)$ $(\lambda x.$`(times n (fct (pred n))))`;
`factorial = fix g;`
`equal` c_6 `(factorial` c_3`);`

5 Tyypitetty lambdakalkyyli

([Pie02] luvut 8 ja 9) Tyyppirelaatio määritellään päättelysäännöillä, jotka liittävät tyyppejä termeille. Tyypitetyssä lambdakalkyylissa jokaiselle termille siis liitetään *tyyppi*, yhdellä termillä voi olla vain yksi tyyppi. Tyyppijärjestelmän tulee olla *turvallinen* eli täyttää etenemis- ja säilymisteoreemat. Yksikään *hyvinmääritelty* termi ei saa "jäädä jumiin" eli termin tulee olla arvo tai sitten siihen on voitava soveltaa jotain evaluaatiosääntöä. Lisäksi hyvinmääriteltävyyden tulee säilyä evaluaatiossa: kun hyvinmääriteltyyn termiin sovelletaan evaluaatioaskel, lopputuloksena saadun termin tulee myös olla hyvinmääritelty.

5.1 Tyyppirelaatio

Tyypitetyssä lambdakalkyylissa on tyypit `Bool` boolean-arvoille ja `Nat` numeroille. Lisäksi funktioille on määritelty numeroituvasti ääretön määrä tyyppejä muotoa $T_1 \to T_2$, joiden tyyppiset funktiot ottavat argumentin tyyppiä T_1 ja palauttavat argumentin muotoa T_2. Abstraktioiden tyypitys voi olla *implisiittistä* tai *eksplisiittistä*. Edellisessä abstraktiolle liitetään tyyppi sen mukaan, minkätyyppisiä sen argumenttien odotetaan olevan. Jälkimmäisessä yritetään päätellä argumentin käytöstä abstraktion vartalossa, mikä tyypin tulisi olla. Argumenttien eksplisiittinen tyypitys ilmaistaan merkinnällä $\lambda x : T_1.t_2$.

Abstraktioiden tyyppirelaatio määritellään sisäkkäisten λ-abstraktioiden vuoksi tyyppirelatio kolmipaikkaiseksi relaatioksi $\Gamma \vdash t : T$, missä Γ on joukko oletuksia t:n vapaiden muuttujien tyypeistä. Tyyppiympäristö Γ laajennetaan oikealle operaattorilla ",", jolloin Γ sitoo muuttujia. Γ:n sitomien muuttujien $dom(\Gamma)$ nimien on erotuttava uusien sidontojen muuttujista, joten sidottuja muuttujia nimetään tarvittaessa uudelleen.

Puhtaan yksinkertaisesti tyypitetyn (ei tyyppejä `Boolean` ja `Nat`) lambdakalkyylin termit, operationaalinen semantiikka ja tyypityssäännöt näkyvät taulukosta 3.

Muuttujien tyyppisääntö kuvaa sen, että muuttuja on mitä tahansa tyyppiä, jota kyseisellä hetkellä oletamme sen olevan. Sääntö T-ABS kuvaa sitä, että abstraktion sitoma muuttuja `x` on tyyppiä T_1 ja oikeanpuolimmainen termi tyyppiä T_2, niin `x`:n ilmentymien termissä t_2 oletetaan olevan tyyppiä T_1, joten funktio t_2 tuottaa tyypin T_1 argumenteilla tuloksen tyyppiä T_2. Säännössä T-ABS premissi lisää johtopäätöksen oletuksiin yhden oletuksen lisää. Laskentasääntö T-APP kuvaa, mitä tyyppioletuksia soveltamisessa tarvitaan. Jos t_1 evaluoituu funktiolle, joka kuvaa tyypin T_{11} argumentit tuloksille tyyppiä T_{12}, ja t_2 evaluoituu tulokseen tyyppiä T_{11}, soveltamisen tulos on tyyppiä T_{12}.

Taulukossa esitettyjen sääntöjen lisäksi yksinkertaisesti tyypitetylle lambdakalkyylille, jossa on tyypit `Boolean` ja `Nat`, käytössä ovat luvussa 8 määritellyt tyypityssäännöt `Boolean`- ja `Nat` -tyypeille. Tyypityssääntöjen avulla voidaan johtaa todistuspuita, joissa esimerkiksi osoitetaan jonkin termin olevan tiettyä typpiä tietyssä

Syntaksi		
`t ::=`		Termit:
	`x`	muuttuja
	λ`x : T. t`	abstraktio
	`t t`	sovellus
`v ::=`		Arvot:
	λ`x : T. t`	abstraktioarvo
`T ::=`		Tyypit:
	`T` $\rightarrow$ `T. t`	funktiotyyppi
Γ `::=`		Kontekstit:
	$\oslash$	tyhjä konteksti
	Γ, `x : T`	termin muuttujien sidonta
Evaluaatio		$t \rightarrow t'$
	$\dfrac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}$	(E-APP1)
	$\dfrac{t_2 \rightarrow t_2'}{v_1 t_2 \rightarrow v_1 t_2'}$	(E-APP2)
	(λ`x : $T_{11}$.` t_{12}) $v_2 \rightarrow [x \rightarrow v_2]t_{12}$	(E-APPABS)
Tyypitys		$t \rightarrow t'$
	$\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)
	$\dfrac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2}$	(T-ABS)
	$\dfrac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(T-APP)

Taulukko 3:

kontekstissa.

5.2 Tyypityksen ominaisuuksia

Taulukossa esitetyn kaltainen puhdas yksinkertaisesti tyypitetty lambdakalkyyli ilman perustyyppejä ei ole hyvintyypitetty, koska siinä ei ole hyvintyypitettyjä termejä eli tyyppiekspressioiden joukko on tyhjä. `Boolean`- ja `Nat`- tyypit sisältävä tyypitetty lambdakalkyyli taas voidaan osoittaa tyyppiturvalliseksi ([Pie02], luku 9.3). Tyyppirelaation määritelmien avulla voidaan osoittaa, että tiettyä syntaktista muotoa olevat termit ovat aina tietyntyyppisiä. Lisäksi termin tyyppi on yksikäsitteinen ja termeihin voidaan aina soveltaa evaluaatiosääntöjä, jos ne eivät ole arvoja. Voidaan osoittaa (sivu 106-107), että tyypit säilyvät myös evaluaation sijoituksessa. Tällöin tyypit säilyvät aina evaluaatiossa. Yksinkertaisesti tyypitetyllä lambdakalkyylilla ei voida ilmaista rekursiota.

Termi `m` tyypittämättömässä lambdakalkyylissa on *tyypitettävä*, jos on olemassa yksinkertaisesti tyypitetty termi `t`, tyyppi `T` ja konteksti Γ, joille $\Gamma \vdash \mathtt{t} : \mathtt{T}$ ja `m` on termin `t` tyypittämätön vastine.

5.3 Curry-Howard-vastaavuus

Curry-Howard-vastaavuus tarkoittaa, että ohjelmointikielet ja logiikka ovat rinnasteisia. Esimerkiksi kaavat vastaavat tyyppejä ja todistukset termejä. Logiikan implikaatiotodistusta voidaan pitää proseduurina, joka muuntaa etujäsenen todistuksen jälkijäsenen todistukseksi, vastaavasti esimerkiksi yksinkertaisen tyypitetyn lambdakalkyylin termi on sen tyyppiä vastaavan loogisen proposition todistus. Laskenta (termien palautus evaluointisäännöillä) vastaa katkaisusäännön soveltamista logiikan todistuksissa. Vastaavuus pätee useille tyyppijärjestelmille ja logiikoille. Yksinkertaista tyypitettyä lambdakalkyylia vastaa lauselogiikka.

Funktiotyypissä ”$\rightarrow$” sääntö T-ABS on tuontisäännön kaltainen ja T-APP eliminointisäännön kaltainen. Edellinen kertoo, kuinka tyypin elementtejä luodaan ja jälkimmäinen kuvaa, kuinka tyypin elementtejä käytetään. Kun tuontisääntö (λ) on eliminointisäännön välitön alitermi, tuloksena on redex.

5.4 Normalisointi

Jokaisen hyvintyypitetyn ohjelman evaluointi päättyy äärellisessä määrässä askelia. Yksinkertaisesti tyypitetyssä lambda-kalkyylissa jokainen hyvinmääritelty termi voidaan normalisoida eli palauttaa normaalimuotoon, jolloin laskenta päättyy. Jos kalkyylia laajennetaan esimerkiksi rekursiolla, tämä ominaisuus ei enää ole voimassa. Siksi normalisointiominaisuus ei yleensä päde täysipainoisille ohjelmointikielille. Kuitenkin tyyppijärjestelmissä tyyppitarkistusalgoritmin suorituksen päättymisen takaa se, että tyyppiekspressioiden normalisointi päättyy. Hyvinmääritelty tyyppijärjestelmä ja tyyppitarkastus sille varmistavat osaltaan ohjelmien oikeellisuutta ja vähentävät ohjelmointivirheitä.

Lähteet

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