Tyypitetty λ-laskenta

Funktionaalisen ohjelmoinnin periaatteet -seminaari

Jan Sjögren

Ohjaaja: Matti Nykänen

Helsingin yliopisto
Tietojenkäsittelytieteen laitos

17. helmikuuta 2000

Sisältö

1 Johdanto **1**

2 Yksinkertaisesti tyypitetty λ-laskenta **1**

2.1 Kontekstinen syntaksi 1

2.2 Kaavat ja tyypit 4

2.3 Tyypintarkistusalgoritmi 5

3 Tyyppimuuttujat ja monimuotoisuus **7**

4 Tyypinpäättely **9**

4.1 Kieli $\lambda_t^{\rightarrow}$ 10

4.2 Sijoitus, ilmentymät ja samastus 10

4.3 Tyypinpäättelyalgoritmi kielelle $\lambda_t^{\rightarrow}$ 12

5 Yhteenveto **14**

Viitteet **14**

1 Johdanto

Tyypittämätön λ-laskenta tarjoaa syntaktisesti yksinkertaisen mallin laskennan kuvaamiseen. Tyypitettyä λ-laskentaa pidetään kuitenkin nykyään hyödyllisempänä ohjelmointikielien analysointivälineenä. Sen avulla voidaan mallintaa tyypitettyjen ohjelmointikielien vahvuudet ja rajoitukset. Lisäksi tyypitetyssä λ-laskennassa vältetään *Russellin paradoksi*, ja myös polymorfisten funktioiden käyttö on mahdollista.

Tässä esityksessä oletetaan tunnetuiksi λ-laskennan perusteet, kuten λ-lausekkeiden syntaksi ja sieventämisen käsite. Tämän pohjalta käsitellään luvussa 2 erästä yksinkertaisesti tyypitetyn λ-laskennan versiota ja siihen liittyvää tyypintarkistusta. Luvussa 3 esitellään lyhyesti monimuotoisuutta tyypitetyssä λ-laskennassa. Luvussa 5 tarkastellaan lopuksi tyypinpäättelyongelmaa.

Tämä esitelmä perustuu pääasiassa J. C. Mitchellin teokseen *Foundations for Programming Languages* [Mit96, luvut 4, 9 ja 11].

2 Yksinkertaisesti tyypitetty λ-laskenta

Yksinkertaisesti tyypitetty λ-laskenta tarkoittaa mitä tahansa tyypitetyn λ-laskennan muotoa, jonka tyypeissä ei esiinny tyyppimuuttujia. Yksinkertaisesti tyypitetyssä λ-laskennassa voidaan tyyppilausekkeissa käyttää tyyppivakioiden (nat, $real$) lisäksi mm. summaa ja karteesista tuloa. Funktiotyypit kuuluvat kaikkiin yksinkertaisesti tyypitetyn λ-laskennan versioihin.

Tyypitetyn λ-laskennan eri versiot voidaan jaotella ja nimetä niissä esiintyvien tyyppien mukaan. Tässä luvussa keskitytään yksinkertaisimpaan näistä kielistä, eli *yksinkertaisesti tyypitettyyn λ-laskentaan funktiotyypeillä (simply typed λ-calculus with function types)*. Tätä kieltä merkitään $\lambda^{\rightarrow}$, ja sen tyyppilausekkeet muodostuvat ainoastaan symbolista $\rightarrow$ ja tyyppivakioista. $\lambda^{\rightarrow}$-tyyppilausekkeet saadaan kieliopista

$$\sigma ::= \ b \mid \sigma \rightarrow \sigma \ ,$$

missä b voi olla mikä tahansa tyyppivakio.

2.1 Kontekstinen syntaksi

Ohjelmointikielet määritellään usein kontekstittomalla kieliopilla. Staattisesti tyypitetyn kielen konstekstiton kuvaaminen ei kuitenkaan kerro kaikkea, koska tyypitysrajoitteet ovat kontekstista riippuvia. Esimerkiksi lausekkeen $x + 3$ tyyppi on hyvinmääritelty ainoastaan, jos x on numeerinen muuttuja. Voidaan myös osoittaa, että ei ole olemassa kontekstitonta kielioppia, joka tuottaisi täsmälleen $\lambda^{\rightarrow}$:n hyvintyypitetyt termit. Tässä luvussa määritellään $\lambda^{\rightarrow}$:n lausekkeet ja niiden tyypit käyttäen aksioomia ja päättelysääntöjä.

Jotta muuttujien tyypit voidaan ottaa huomioon päättelysäännöissä, otetaan käyttöön *tyypitysväite (typing assertion)*

$\Gamma \rhd M : \tau$,

missä Γ on *tyypityskonteksti (typing context, type assignment)* muotoa

$\Gamma = \{x_1 : \sigma_1, \ldots, x_k : \sigma_k\},\ x_i = x_j\ joss\ i = j.$

Muuttuja voi siis esiintyä kontekstissa ainoastaan kerran.

Intuitiivisesti tyypitysväite $\Gamma \rhd M : \tau$ tarkoittaa, että jos muuttujat $x_1, \ldots, x_k$ ovat tyypeiltään $\sigma_1, \ldots, \sigma_k$, niin M on hyvinmääritelty tyypin τ termi. $\Gamma \rhd M : \tau$ voidaan lausua "kontekstissa Γ termin M tyyppi on τ".

Mikäli Γ on jokin tyypityskonteksti, merkitsee $\Gamma, x : \sigma$ tyypityskontekstia, jolle on voimassa

$\Gamma, x : \sigma = \Gamma \cup x : \sigma$.

Tällaisessa tapauksessa oletetaan, ettei x esiinny Γ:ssa.

$\lambda^{\rightarrow}$:n atomiset lausekkeet saadaan tyypitysaksioomista. Tyypitysaksiooma $c : \tau$ merkitsee intuitiivisesti, että symboli c on tyyppiä τ. Toisin sanoen $c : \tau$ on tyyppiin kuulumista koskeva aksiooma, joka tässä esityksessä merkitään ":". Aksiooman mukaan c:n ja τ:n välillä on siis voimassa tyyppinkuulumisrelaatio.

Jokaista tyypin σ vakiota c vastaa aksiooma

(cst) $\qquad \emptyset \rhd c : \sigma$.

Tyypityskonteksti on tässä tapauksessa tyhjä, koska vakion tyyppi on kiinteä ja siksi riippumaton kontekstista.

Oletetaan käytössä olevan numeroituvasti äärettömän muuttujajoukon $v_0, v_1, v_2, \ldots$. Muuttujat saavat tyypit aksioomalla

(var) $\qquad x : \sigma \rhd x : \sigma$,

missä σ:n täytyy olla $\lambda^{\rightarrow}$-tyyppi. Tämä aksiooma merkitsee yksinkertaisesti sitä, että muuttujalla x on juuri se tyyppi, joka sille on tyypityskontekstissa määritelty.

Rakenteiset lausekkeet ja niiden tyypit määritellään käyttämällä päättelysääntöjä. Nämä säännöt tekevät mahdolliseksi monimutkaisempien asioiden johtamisen ":"-relaatiosta. Sääntö

(add var)

$$\frac{\Gamma \rhd M : \sigma}{\Gamma, x : \tau \rhd M : \sigma}$$

sallii ylimääräisen hypoteesin lisäämisen tyypityskontekstiin. Siis jos M on tyyppiä σ kontekstissa Γ, niin M on tyyppiä σ kontekstissa $\Gamma, x : \tau$, joka antaa tyypin myös x:lle.

λ-laskennassa λ-abstraktiota käytetään osoittamaan muuttuja, jota käytetään funktion argumenttina. Tämä ilmaistaan täsmällisesti seuraavalla säännöllä.

($\rightarrow$ Intro)

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd (\lambda x : \sigma.M) : \sigma \rightarrow \tau}.$$

Tämä sääntö merkitsee, että mikäli M on tyyppiä τ aina, kun $x : \sigma$ kuuluu tyypityskontekstiin, niin lauseke $\lambda x : \sigma.M$ määrittelee funktion tyyppiä $\sigma \rightarrow \tau$.

Funktion soveltamiset muodostetaan säännöllä

($\rightarrow$ Elim)

$$\frac{\Gamma \rhd M : \sigma \rightarrow \tau, \qquad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \tau},$$

joka merkitsee, että tyyppiä $\sigma \rightarrow \tau$ olevaa funktiota voidaan soveltaa tyyppiä σ olevaan argumenttiin ja tällöin saadaan tyyppiä τ oleva tulos.

Esimerkki 1. Eräs tapaus, jonka johtaminen vaatii kaikkien edellä esiteltyjen sääntöjen käyttöä, on $y : \sigma \rhd (\lambda x : \sigma.x)y : \sigma$. Aloitetaan aksioomalla $x : \sigma \rhd x : \sigma$ ja sovelletaan sitten sääntöä ($\rightarrow$ Intro).

$$\frac{x : \sigma \rhd x : \sigma}{\emptyset \rhd \lambda x : \sigma.x : \sigma \rightarrow \sigma}$$

Seuraavaksi sovelletaan sääntöä (add var), ja saadaan

$$\frac{\emptyset \rhd \lambda x : \sigma.x : \sigma \rightarrow \sigma}{y : \sigma \rhd \lambda x : \sigma.x : \sigma \rightarrow \sigma}.$$

Nyt konteksti on oikea. Yhdistetään nyt edellinen tulos aksioomaan $y : \sigma \rhd y : \sigma$ säännöllä ($\rightarrow$ Elim) halutun lopputuloksen aikaansaamiseksi.

$$\frac{y : \sigma \rhd \lambda x : \sigma.x : \sigma \rightarrow \sigma, \qquad y : \sigma \rhd y : \sigma}{y : \sigma \rhd (\lambda x : \sigma.x)y : \sigma} \quad \square$$

2.2 Kaavat ja tyypit

Tarkastelemalla tyypitetyn λ-laskennan ja konstruktiivisen logiikan kaavojen suhdetta voidaan havaita kiinnostavia yhtäläisyyksiä. Seuraavassa esitellään lyhyesti tämä aihealue.

Implikaatiologiikka (implicational logic) on propositiologiikkaa, jonka kaavat sisältävät ainoastaan propositiomuuttujia sekä implikaation (merkitään tässä symbolilla $\rightarrow$). Tämän logiikan kaavat voidaan määritellä kieliopilla

$\sigma ::= \; b \mid \sigma \rightarrow \sigma$,

missä b voi olla mikä tahansa propositiomuuttuja. Tarkastellaan seuraavaksi *luonnollista päättelyä (natural deduction)* implikaatiologiikassa. Se on todistusjärjestelmä, joka pyrkii jäljittelemään tyypillistä "liitutaulutodistamista". Alla esitetään epämuodollisesti tällaisen todistuksen muoto.

Oletetaan σ.

Askelten ... jälkeen päädytään tulokseen τ.

Tämän perusteella nähdään, että $\sigma \rightarrow \tau$.

Tässä lähdetään siis liikkeelle oletuksesta σ, ja johdetaan siitä τ. Lopuksi todetaan, että koska σ:n avulla voidaan todistaa τ todistamiseen, on voimassa implikaatio $\sigma \rightarrow \tau$.

Luonnollisen päättelyn todistusjärjestelmä implikaatiologiikalle voidaan esittää käyttäen yhtä aksioomaa ja kolmea päättelysääntöä, jotka esitetään alla. Säännöissä merkitään Γ:lla äärellistä joukkoa nimettyjä oletuksia. Jos x on oletuksen nimi ja σ kaava, niin x:ksi nimetty oletus σ voidaan kirjoittaa $x : \sigma$. Merkintä $\Gamma \vdash_M \sigma$ tarkoittaa, että M on kaavan σ todistus, joka perustuu nimettyjen oletuksien joukkoon Γ.

$$x : \sigma \vdash_x \sigma$$

$$\frac{\Gamma \vdash_M \sigma}{\Gamma, x : \tau \vdash_M \sigma} \quad x \text{ ei esiinny } \Gamma\text{:ssa}$$

$$\frac{\Gamma \vdash_M \sigma \to \tau, \qquad \Gamma \vdash_N \sigma}{\Gamma \vdash_{MN} \tau}$$

$$\frac{\Gamma, x : \sigma \vdash_M \tau}{\Gamma \vdash_{\lambda x:\sigma.M} \sigma \to \tau}$$

Voidaan havaita, että tämä määrittely vastaa tyypitettyä λ-laskentaa: tyypitetyn λ-laskennan termit ovat täsmälleen edellä määritellyt todistukset ja termien tyypit vastaavat todistettuja kaavoja. Voidaan siis merkitä $\Gamma \vdash_M \sigma$ joss $\Gamma \rhd M : \sigma$. Lisäksi todistuksen normalisointi vastaa tyypitettyjen λ-termien sievennystä. *Kaavat tyyppeinä -vastaavuus (formulas-as-types correspondence)* on siis kolmiosainen:

Kaava	$\approx$	tyyppi
Todistus	$\approx$	λ-termi
Todistuksen normalisointi	$\approx$	sievennys

Tätä vastaavuutta kutsutaan myös *Curry-Howard isomorfismiksi*. Aihetta käsitellään tarkemmin mm. Hindleyn teoksessa *Basic Simple Type Theory* [Hin97, luku 6].

2.3 Tyypintarkistusalgoritmi

Yksinkertaisesti tyypitettyyn λ-laskentaan liittyy useita algoritmisia ongelmia. Voidaan esimerkiksi haluta selvittää tehokkaasti, onko $\Gamma \rhd M : \sigma$ voimassa, jos Γ, M ja σ tunnetaan. Toisaalta voidaan haluta selvittää, onko olemassa sellainen σ, jolla $\Gamma \rhd M : \sigma$ on voimassa, kun Γ ja M tunnetaan. Tässä luvussa esitetään algoritmi, joka ratkaisee jälkimmäisen näistä ongelmista.

$\lambda^{\to}$:n lausekkeet saadaan kontekstittomasta kieliopista

$M ::= \ c \mid x \mid MM \mid \lambda x : \sigma.M$,

missä c voi olla mikä tahansa vakio ja σ voi olla mikä tahansa $\lambda^{\to}$-tyyppilauseke. Tämän kieliopin tuottamia lausekkeita kutsutaan $\lambda^{\to}$:n *esitermeiksi (preterm)*.

Tyypintarkistusalgoritmi esitetään taulukossa 1. Algoritmi saa syötteekseen tyypityskontekstin Γ ja esitermin M ja joko palauttaa tyypin σ, jolla on voimassa $\Gamma \rhd M : \sigma$, tai pysähtyy epäonnistumiseen.

$$
\begin{array}{rcl}
TC(\Gamma, c) & = & \sigma \\
 & & \text{jos } c : \sigma \\
 & & \text{muuten } \mathit{epäonnistuminen} \\
TC(\Gamma, x) & = & \sigma \\
 & & \text{jos } x : \sigma \in \Gamma \\
 & & \text{muuten } \mathit{epäonnistuminen} \\
TC(\Gamma, MN) & = & \tau \\
 & & \text{jos } TC(\Gamma, M) = \sigma \to \tau \text{ ja } TC(\Gamma, N) = \sigma \\
 & & \text{muuten } \mathit{epäonnistuminen} \\
TC(\Gamma, \lambda x : \sigma.M) & = & \sigma \to \tau \\
 & & \text{jos } TC((\Gamma_{\lambda x:\sigma.M}, x : \sigma), M) = \tau \\
 & & \text{muuten } \mathit{epäonnistuminen}
\end{array}
$$

Taulukko 1: Tyypintarkistusalgoritmi TC.

λ-abstraktion käsittelevässä algoritmin haarassa käytetään merkintää, joka määritellään seuraavasti:

$\Gamma_\phi = \{x : \tau \in \Gamma \mid x$ esiintyy vapaana termissä $\phi\}$.

Esimerkki 2. Käytetään algoritmia TC selvittämään termin $\lambda x : nat \to nat.\lambda y : nat.xy$ tyyppi. Alkutilanteesta

$TC(\emptyset, \lambda x : nat \to nat.\lambda y : nat.xy)$

päädytään laskemaan

$TC(\{x : nat \to nat\}, \lambda y : nat.xy)$,

jonka selvittämiseksi tarvitaan

$TC(\{x : nat \to nat, y : nat\}, xy)$.

Koska $TC(\{x : nat \to nat, y : nat\}, x) = nat \to nat$
ja $TC(\{x : nat \to nat, y : nat\}, y) = nat$, niin

$TC(\{x : nat \to nat, y : nat\}, xy) = nat$,

minkä perusteella

$TC(\{x : nat \to nat\}, \lambda y : nat.xy) = nat \to nat$.

Tulokseksi saadaan siis

$TC(\emptyset, \lambda x : nat \to nat.\lambda y : nat.xy) = (nat \to nat) \to (nat \to nat)$. □

3 Tyyppimuuttujat ja monimuotoisuus

Staattisella tyypityksellä on yksinkertaisesti tyypitetyssä λ-laskennassa ja sitä vastaavissa ohjelmointikielissä erilaisia tarkoituksia. Se esimerkiksi estää tyyppivirheet. Yksinkertaisesti tyypitetyn λ-laskennan tyypitysrajoitteilla on kuitenkin joitakin haittapuolia. On nimittäin olemassa monia käyttökelpoisia lausekkeita, joiden tyyppi ei ole hyvinmääritelty.

Tyypillinen esimerkki yksinkertaisen tyyppirakenteen aiheuttamista hankaluuksista on järjestämisfunktioiden käsittely Pascalin tapaisessa kielessä. Järjestämisongelmassa on tavallisesti joukko alkioita, joista jokaisessa on avain. Avaimen tyypillä ei ole väliä, kunhan käytössä on vertailufunktio, jolla pystytään selvittämään kahden avaimen keskinäinen järjestys. Itse järjestämisalgoritmi ei riipu mitenkään avaimen tai alkion tyypistä, joten pitäisi olla mahdollista laatia funktio, joka pystyy järjestämään monentyyppisiä alkioita, jos se saa parametrikseen vastaavan vertailufunktion. Tämä ei kuitenkaan onnistu Pascalin kaltaisessa kielessä, vaan jokaiselle tietotyypille tarvitaan erillinen funktio, mikä aiheuttaa monenlaisia hankaluuksia. Tämän vuoksi voidaan haluta käyttää kieltä, joka sallii samaa funktiota sovellettavan erityyppisiin tietoihin.

Yksi ratkaisumalli ongelmaan voidaan esittää käyttäen funktion yhdistämistä, joka on hieman yksinkertaisempi tapaus kuin järjestäminen. $\lambda^{\rightarrow}$-funktio

$$compose_{nat,nat,nat} =_{def} \lambda f : nat \rightarrow nat.\lambda g : nat \rightarrow nat.\lambda x : nat.f(gx)$$

yhdistää kaksi luonnollisia lukuja käsittelevää funktiota. Tyyppiä $real \rightarrow real$ olevia funktioita taas voidaan yhdistää funktiolla

$$compose_{real,real,real} =_{def} \lambda f : real \rightarrow real.\lambda g : real \rightarrow real.\lambda x : real.f(gx)\ .$$

Nämä funktiot eroavat toisistaan selvästikin vain parametrien tyyppien osalta. Koska molemmat funktiot laskevat tuloksensa täsmälleen samalla tavalla, niiden voitaisiin olettaa olevan jonkin yleisen yhdistämisfunktion "ilmentymiä".

Polymorfinen funktio on funktio, jota voidaan soveltaa monentyyppisiin argumentteihin. $\lambda^{\rightarrow}$ voidaan laajentaa sisältämään polymorfiset funktiot ulottamalla λ-abstraktio tyyppimuuttujiin. Käyttämällä tyyppimuuttujia r, s ja t voidaan kirjoittaa "yleinen" yhdistämisfunktio

$$compose_{r,s,t} =_{def} \lambda f : s \rightarrow t.\lambda g : r \rightarrow s.\lambda x : r.f(gx)\ .$$

$compose_{r,s,t}$-funktion tyyppi on siis $(s \rightarrow t) \rightarrow (r \rightarrow s) \rightarrow (r \rightarrow t)$. Siis kun kaikki kolme tyyppimuuttujaa tulkitaan luonnollisten lukujen tyypiksi,

$compose_{r,s,t}$:n merkitys on sama kuin $compose_{nat,nat,nat}$:n. Soveltamalla λ-abstraktiota tyyppimuuttujiin r, s ja t voidaan kirjoittaa funktio, joka tuottaa funktion $compose_{nat,nat,nat}$, kun sitä sovelletaan argumentteihin nat, nat ja nat. Tyyppimuuttujien erottamiseksi tavallisista muuttujista otetaan käyttöön symboli T merkitsemään jotakin tyyppien joukkoa, joka sisältää ainakin kaikki $\lambda^{\rightarrow}$:n tyypit.

Käyttämällä λ-abstraktiota tyyppeihin voidaan määritellä polymorfinen yhdistämisfunktio

$$compose =_{def} \lambda r : T.\lambda s : T.\lambda t : T.compose_{r,s,t} \; .$$

Tämä funktio vaatii kolme tyyppiparametria, ennenkuin sitä voidaan soveltaa tavalliseen funktioon. Jos *compose*-funktiota halutaan soveltaa kahteen numeeriseen funktioon $f, g : nat \rightarrow nat$, niin sitä sovelletaan ensin tyyppiargumentteihin nat, nat ja nat. Käyttämällä β-sievennystä saadaan

$$compose\ nat\ nat\ nat$$

$$= (\lambda r : T.\lambda s : T.\lambda t : T.\lambda f : s \rightarrow t.\lambda g : r \rightarrow s.\lambda x : r.f(gx))\ nat\ nat\ nat$$

$$= \lambda f : nat \rightarrow nat.\lambda g : nat \rightarrow nat.\lambda x : nat.f(gx) \; .$$

Siis $compose\ nat\ nat\ nat = compose_{nat,nat,nat}$.

Jos tyypitettyyn λ-laskentaan kuuluu tyyppiabstraktio ja -soveltaminen, myös polymorfisille funktioille täytyy määritellä tyyppi. Tarkastellaan funktiota

$$Id =_{def} \lambda t : T.\lambda x : t.x \; .$$

Funktion Id lähtöjoukko on selvästikin T. Koska $Idt : t \rightarrow t$ kaikilla $t \in T$, niin Id-funktion voidaan ajatella kuuluvan äärettömään tuloon $\Pi_{t:T} t \rightarrow t$. Tämän perusteella otetaan käyttöön merkintä $Id : \Pi t.t \rightarrow t$.

Tyyppisoveltamisen tyyppi voidaan määrittää polymorfisen funktion tyypin ja tyyppiargumentin perusteella samaan tapaan kuin tavallisenkin soveltamisen tyyppi. Esimerkiksi soveltamisen $Id\tau$ tyyppi saadaan sijoittamalla τ sidottuun muuttujaan t tyypissä $t \rightarrow t$. Vaikka tämä sijoitus saattaa näyttää jonkinlaiselta β-sievennykseltä (jossa on siis λ:n paikalla Π), tämä sijoitus tehdään kuitenkin ainoastaan Id:n tyyppiin, eikä funktioon itseensä. Tämä merkitsee, että polymorfisten funktioiden soveltamisten tyypit voidaan tarkistaa evaluoimatta soveltamista.

Aiemmin kun T otettiin käyttöön, polymorfisten funktioiden tyyppejä ei vielä ollut käytössä. On kuitenkin päätettävä, mitä tyyppejä T tarkalleen sisältää. Mikäli joukkoon T kuuluu ainoastaan kokoelma tyyppivakioita sekä yksinkertaiset tyypit, jotka voidaan määritellä kielen tyyppivakioista, kyseessä on semanttisesti suoraviivainen laskenta, jota kutsutaan *predikatiiviseksi monimuotoisuudeksi (predicative polymorphism)*. "Predikatiivinen" viittaa siihen, että T otetaan käyttöön vasta, kun kaikki sen alkiot on määritelty. T ei siis sisällä sellaisia tyyppejä, joiden määritelmässä esiintyy T. On myös mahdollista määritellä, että T sisältää kaikki polymorfiset tyypit, tai käsitellä myös joukkoa T tyyppinä ja antaa sen sisältää kaikki tyypit mukaanlukien itsensä. Näistä vaihtoehdoista saatavia monimuotoisuuden lajeja ei kuitenkaan käsitellä tässä yhteydessä tarkemmin.

4 Tyypinpäättely

Tyypinpäättely (type inference) tarkoittaa tyypittämättömien tai "osittain tyypitettyjen" lausekkeiden muuntamista hyvintyypitetyiksi termeiksi päättelemällä puuttuva tyyppi-informaatio. Tämä on paitsi teoreettisesti kiinnostava, myös käytännönläheinen kysymys. Jos nimittäin halutaan toteuttaa ohjelmointikieli, jolle voidaan tehdä käännösaikainen tyyppitarkistus, mutta jossa ei tarvitse antaa täydellistä tyyppitietoa kaikista muuttujista, voidaan käyttää tyypinpäättelyä lisäämään tyyppi-informaatio automaattisesti.

Yksi tapa esittää tyypinpäättelyongelma yleisellä tasolla on määritellä *poistofunktio* $Erase$ tyypitetyltä kieleltä $\mathcal{L}$ jollekin toiselle kielelle $\mathcal{L}'$. Tavallisesti $\mathcal{L}'$ on vastaava tyypittämätön kieli tai sitten sellainen versio $\mathcal{L}$:stä, josta puuttuu osa tyyppi-informaatiosta.

Esimerkki 3. Poistofunktio $\lambda^{\rightarrow}$:n termeiltä tyypittämättömille λ-termeille poistaa tyyppimääritykset λ-abstraktioista. Määritellään $Erase$ seuraavasti:

$$\begin{aligned} Erase(c) &= c \\ Erase(x) &= x \\ Erase(\lambda x : \tau . M) &= \lambda x . Erase(M) \\ Erase(MN) &= Erase(M) Erase(N) \qquad \square \end{aligned}$$

Tyypinpäättelyongelma voidaan nyt muotoilla seuraavasti kielelle $\mathcal{L}$ ja poistofunktiolle $Erase : \mathcal{L} \rightarrow \mathcal{L}'$.

Tyypinpäättelyongelma: Kun tunnetaan $U \in \mathcal{L}'$, niin etsi sellainen kielen $\mathcal{L}$ tyypitetty termi $\Gamma \triangleright M : \tau$, jolla $Erase(M) = U$.

Yleisesti ottaen voi olla äärettömän monta tapaa lisätä tyyppi-informaatio termeihin. Esimerkiksi termille $\lambda f . \lambda x . f(fx)$ voitaisiin antaa mikä tahansa muotoa $(\tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$ oleva tyyppi.

4.1 Kieli $\lambda_t^{\to}$

Tässä luvussa keskitytään tyypinpäättelyyn kielelle $\lambda_t^{\to}$, joka saadaan lisäämällä kieleen $\lambda^{\to}$ tyyppimuuttujat. $\lambda_t^{\to}$:n tyypit voidaan muodostaa kieliopilla

$\tau ::= \; b \mid t \mid \tau \to \tau$,

missä b on tyyppivakio ja t tyyppimuuttuja. $\lambda_t^{\to}$:n termit ovat muotoa

$M ::= \; c \mid x \mid \lambda x : \tau.M \mid MM$.

$\lambda_t^{\to}$:n tyypitysaksioomat ja päättelysäännöt ovat samat kuin $\lambda^{\to}$:lla, paitsi että tyyppilausekkeet ja tyypityskontekstit saavat sisältää tyyppimuuttujia.

Jatkossa tutkitaan tyypinpäättelyongelmaa, joka perustuu esimerkin *Erase*-funktioon kuvauksena $\lambda_t^{\to}$-termeiltä tyypittämättömille λ-termeille. Tätä kutsutaan *Curryn tyypinpäättelyongelmaksi* (*Curry type inference problem*).

On kuitenkin muistettava, että kaikille termeille ei voida määrittää tyyppiä. Tyypittämätön termi U voidaan nimittäin tyypittää vain, jos ei ole olemassa päättymätöntä sarjaa U:n sievennyksiä. Esimerkki termistä, joka ei normalisoidu ja jota ei siis voida tyypittää, on $(\lambda x.xx)(\lambda x.xx)$.

4.2 Sijoitus, ilmentymät ja samastus

Tyyppisijoitus (type substitution) on funktio tyyppimuuttujilta tyyppilausekkeille. Jos τ on tyyppilauseke ja S on tyyppisijoitus, niin $S\tau$ on tyyppilauseke, joka saadaan korvaamalla jokainen muuttuja t lausekkeessa τ $S(t)$:llä. Tyyppisijoitusta S voidaan soveltaa tyyppikontekstiin Γ, jolloin pätee

$S\Gamma = \{x : S\tau \mid x : \tau \in \Gamma\}$.

Jos tyyppisijoitusta S sovelletaan $\lambda_t^{\to}$-termiin M, tulos SM eroaa M:stä ainoastaan sidottujen muuttujien osalta. Tyyppisijoituksen avulla voidaan muotoilla seuraava määritelmä.

Määritelmä 1. Tyypitysväite $\Gamma' \rhd M' : \tau'$ on tyypitysväitteen $\Gamma \rhd M : \tau$ *ilmentymä (instance)*, jos on olemassa sellainen tyyppisijoitus S, että

$\Gamma' \supseteq S\Gamma$, $M' = SM$ ja $\tau' = S\tau$.

Tärkeä osa tyypinpäättelyä on *samastus (unification)*. Jos E on joukko tyyppilausekkeiden välisiä yhtälöitä, niin tyyppisijoitus S samastaa E:n jos $S\tau \equiv S\rho$ kaikille yhtälöille $\tau = \rho \in E$. Toisin sanoen samastuksessa etsitään tyyppisijoitus, joka tekee jokaisesta yhtälöstä syntaktisesti toden. Esimerkiksi joukon $\{s = t \to v, t = v \to w\}$ samastin on sijoitus, joka korvaa t:n tyypillä $v \to w$ ja s:n tyypillä $(v \to w) \to v$.

$Unify(\emptyset) = \emptyset$

$Unify(E \cup \{b_1 = b_2\}) =$
  jos $b_1 \not\equiv b_2$ niin *epäonnistuminen*
    muuten $Unify(E)$

$Unify(E \cup \{t = \tau\}) =$
  jos $t \equiv \tau$ niin $Unify(E)$
    muuten jos t esiintyy τ:ssa niin *epäonnistuminen*
      muuten $Unify([\tau/t]E) \circ [\tau/t]$

$Unify(E \cup \{\sigma_1 \to \sigma_2 = \tau_1 \to \tau_2\}) =$
  $Unify(E \cup \{\sigma_1 = \tau_1, \sigma_2 = \tau_2\})$

Taulukko 2: Samastusalgoritmi *Unify*.

Algoritmi samastuksen suorittamiseen on esitetty taulukossa 2. Sen toimintaperiaatteena on rekursiivisesti hajottaa osiin "samanmuotoisten" rakenteisten tyyppien yhtälöt sijoittaen tyyppejä tyyppimuuttujiin tarpeen mukaan. Algoritmi päättyy epäonnistumiseen, jos käy tarpeelliseksi samastaa kaksi eri tyyppivakiota tai jos "esiintymistarkistus" onnistuu. Tämä johtuu siitä, että yhtälöille kuten $t = t \to t$ ei koskaan löydetä ratkaisuja, koska käytössä ei ole rekursiivisia tyyppejä. Sen vuoksi aina ennen kuin tyyppimuuttuja korvataan tyypillä algoritmi tarkistaa, ettei muuttuja esiinny tyypissä.

Algoritmi $Unify$ on epädeterministinen, mutta sen tulos on deterministinen. Tulos on siis aina sama, vaikka suoritusjärjestys olisikin erilainen. Tulos on myös yleisin samastava tyyppisijoitus, kun *sijoitus S on yleisempi kuin sijoitus R*, jos on olemassa sijoitus T, jolle pätee $R = T \circ S$. Intuitiivisesti kaikkein yleisin tyyppisijoitus, joka samastaa joukon E, on yksinkertaisin tapa saada kaikki E:n yhtälöt syntaktisesti tosiksi.

$$
\begin{array}{rcl}
PT(c) & = & \emptyset \rhd c : \tau, \text{ missä } \tau \text{ on } c\text{:n muuttujaton tyyppilauseke} \\[1em]
PT(x) & = & \{x : t\} \rhd x : t \\[1em]
PT(UV) & = & \textbf{let} \\
 & & \quad \Gamma \rhd M : \tau = PT(U) \\
 & & \quad \Gamma' \rhd N : \rho = PT(V), \\
 & & \qquad \text{missä tyyppimuuttujat on uudelleennimetty} \\
 & & \quad S = Unify(\{\alpha = \beta \mid x : \alpha \in \Gamma \text{ ja } x : \beta \in \Gamma'\} \cup \{\tau = \rho \to t\}), \\
 & & \qquad \text{missä } t \text{ on uusi tyyppimuuttuja} \\
 & & \textbf{in} \\
 & & \quad S\Gamma \cup S\Gamma' \rhd S(MN) : St \\[1em]
PT(\lambda x.U) & = & \textbf{let } \Gamma \rhd M : \rho = PT(U) \\
 & & \textbf{in} \\
 & & \quad \textbf{if } x : \tau \in \Gamma \text{ jollekin } \tau \\
 & & \qquad \textbf{then } \Gamma - \{x : \tau\} \rhd \lambda x : \tau.M : \tau \to \rho \\
 & & \qquad \textbf{else } \Gamma \rhd \lambda x : s.M : s \to \rho \\
 & & \qquad\quad \text{missä } s \text{ on uusi tyyppimuuttuja}
\end{array}
$$

Taulukko 3: Algoritmi PT ensisijaisen tyypityksen selvittämiseen.

4.3 Tyypinpäättelyalgoritmi kielelle $\lambda_t^{\to}$

Jos U on tyypittämätön λ-termi, niin *U:n eksplisiittinen tyypitys (explicit typing)* on tyypitetty termi $\Gamma \rhd M : \tau$, jolle pätee $Erase(M) = U$. Tyypittämättömällä termillä voi olla useita eksplisiittisiä tyypityksiä. $\lambda_t^{\to}$:ssa on kuitenkin jokaisella tyypitettävissä olevalla termillä on *ensisijainen (principal)* eksplisiittinen tyypitys. Tämä on sellainen eksplisiittinen tyypitys, jota voidaan pitää termin kaikkien mahdollisten tyypitysten edustajana. Täsmällisemmin sanottuna U:n eksplisiittinen tyypitys $\Gamma \rhd M : \tau$ on ensisijainen, jos jokainen U:n eksplisiittinen tyypitys on termin $\Gamma \rhd M : \tau$ ilmentymä.

Taulukossa 3 esitetään algoritmi PT, joka ottaa syötteekseen tyypittämättömän termin U ja joko tuottaa U:n ensisijaisen eksplisiittisen tyypityksen tai päättyy epäonnistumiseen.

Algoritmin toiminta on suoraviivaista vakion kohdalla, koska vakiolla voi olla ainoastaan yksi tyyppi (johon ei sisälly tyyppimuuttujia). Yleisin asia, joka muuttujasta x voidaan todeta on, että jos oletuksissa on $x : \tau$, niin muuttuja

x on tyyppiä τ. Tämä esitetään tyypityksellä $x : \tau \rhd x : \tau$, koska jokainen muotoa $\Gamma, x : \tau \rhd M : \tau$ oleva tyypitysväite on sen ilmentymä.

λ-abstraktion tapaus on suhteellisen yksinkertainen. Jos tiedetään, että λ-abstraktion U vartalolla on tyypitys $\Gamma \rhd M : \rho$, niin on luonnollista antaa termille $\lambda x.U$ tyypitys $\Gamma' \rhd M : \tau \to \rho$, missä $\Gamma = \Gamma', x : \tau$. Algoritmi tarkistaa myös sen mahdollisuuden, että Γ ei olekaan muotoa $\Gamma', x : \tau$. Jos x ei esiinny Γ:ssa, niin x ei voi esiintyä vapaana U:ssa, ja tällöin funktiota voidaan soveltaa minkätyyppiseen argumenttiin tahansa.

Soveltaminen on monimutkaisin tapaus. Se tuottaa tyypityksen ainoastaan, jos samastus onnistuu, ja pysähtyy muuten epäonnistumiseen. Kun U:n tyypitys $\Gamma \rhd M : \tau$ ja V:n tyypitys $\Gamma' \rhd M : \rho$ on selvitetty, täytyy löytää tyyppisijoitus, jonka avulla niiden tyypityskontekstit voidaan yhdistää ja joka antaa termeille asianmukaiset tyypit. Kontekstit voidaan yhdistää etsimällä sellainen sijoitus S, että $S\Gamma \cup S\Gamma'$ antaa kullekin muuttujalle korkeintaan yhden tyypin. Täytyy myös pitää huolta siitä, että $S\tau$ on funktiotyyppi muotoa $S\rho \to \sigma$. Nämä rajoitteet toteutuvat joss on olemassa tyyppisijoitus

$$S = Unify(\{\alpha = \beta \mid x : \alpha \in \Gamma \text{ ja } x : \beta \in \Gamma'\} \cup \{\tau = \rho \to t\}) ,$$

jossa t on uusi tyyppimuuttuja. Algoritmissa oletetaan, että mikäli samastus ei onnistu, sen tulos on määrittelemätön ja koko algoritmi pysähtyy epäonnistumiseen.

Esimerkki 4. Määritetään algoritmilla PT termin $\lambda x.\lambda y.xy$ ensisijainen tyypitys. Rekursiivisia kutsuja seuraamalla nähdään, että ensin tarvitaan tyypitystä $PT(\lambda y.xy)$ ja sitten tyypitystä $PT(xy)$, jolloin päädytään tilanteeseen

$$\begin{array}{lll} PT(xy) & = & \textbf{let} \\ & & \quad x : r \rhd x : r = PT(x) \\ & & \quad y : s \rhd y : s = PT(y), \\ & & \quad S = Unify(\{r = s \to t\}), \\ & & \textbf{in} \\ & & \quad \{x : Sr\} \cup \{y : Ss\} \rhd xy : St. \end{array}$$

Koska S yksinkertaisesti korvaa r:n tyyppilausekkeella $s \to t$ (siis $S = [s \to t/r]$), niin saadaan

$$PT(XY) = \{x : s \to t, y : s\} \rhd xy : t$$

ja edelleen

$$PT(\lambda x.\lambda y.xy) = \emptyset \rhd \lambda x : s \to t.\lambda y : s.xy : (s \to t) \to s \to t\ . \ \square$$

Esimerkki 5. Tarkastellaan, miten algoritmi PT päättyy epäonnistumiseen termillä $\lambda x.xx$. $PT(\lambda x.xx)$:n laskennassa tarvitaan tyypitystä $PT(xx)$, eli

$$
\begin{array}{rcl}
PT(xx) & = & \textbf{let} \\
 & & \quad x : r \rhd x : r = PT(x) \\
 & & \quad x : s \rhd x : s = PT(x) \\
 & & \quad S = Unify(\{r = s\} \cup \{r = s \to t\}), \\
 & & \textbf{in} \\
 & & \quad \{x : Sr\} \cup \{x : Ss\} \rhd xx : St.
\end{array}
$$

Nyt samastus on selvästikin mahdotonta, joten $PT(\lambda x.xx)$ päättyy epäonnistumiseen kutsussa $PT(xx)$. Tyypittämättömälle λ-termille $\lambda x.xx$ ei siis ole mahdollista antaa tyyppiä. $\square$

5 Yhteenveto

Tyypitetty λ-laskenta ja tyypitetyt ohjelmointikielet tarjoavat selviä etuja verrattuna niiden tyypittämättömiin versioihin. Toisaalta heikommin tyypitetyt kielet voivat olla käytännössä miellyttävämpiä työvälineitä. Automaattinen tyypinpäättely antaa mahdollisuuden yhdistää molempien mallien parhaat puolet. On kuitenkin syytä muistaa, että käytettäessä tyypittämätöntä laskentaa ja tyypinpäättelyä voidaan jälleen kohdata ongelmia, jotka alunperin ratkaistiin ottamalla käyttöön tyypit.

Viitteet

[Hin97] Hindley, J. R., Basic Simple Type Theory. Cambridge University Press, 1997.

[Mit96] Mitchell, J. C., Foundations for Programming Languages. MIT Press, 1996.