Riippuvat tyypit ja konstruktiivinen tyyppiteoria

Pekka Anton Uronen

15 maaliskuuta 2000

Funktionaalisen ohjelmoinnin periaatteet -seminaari
Ohjaaja Matti Nykänen

Ohjelmointikielen tyyppijärjestelmän tehtävä on auttaa ohjelmoijaa kirjoittamaan toimiva ohjelma. Tyyppijärjestelmä asettaa ohjelmille järkeviä rajoituksia, joiden rikkomiset kääntäjä voi havaita. Kun tyyppijärjestelmää rikastetaan yhä joustavammaksi, tullaan lopulta tilanteeseen, jossa funktion tyyppi voi riippua sen argumenttien arvoista. Jos sallitaan tyyppien mielivaltaiset riippuvuudet arvoista, voidaan koko ohjelman spesifikaatio koodata tyypiksi. Riippuvat tyypit ovat olennainen osa konstruktiivista tyyppiteoriaa, joka on eräs konstruktiivisen matematiikan formalisointi.

"Going from Hindley-Milner to dependent types is a bit like going from C to Haskell."

Lars Lundgren

1 Johdanto

Vahva tyypitys on osoittautunut arvokkaaksi avuksi oikein toimivien ohjelmien kirjoittamisessa. Staattisesti tyypitetyn kielen kääntäjä tarkistaa käännösvaiheessa, että ohjelman funktiot ja muut operaatiot saavat argumenteikseen ainoastaan sallitun tyyppisiä arvoja. Dynaamisesti tyypitetyssä kielessä tarkistus tehdään vasta ohjelmaa ajettaessa. Tehtiin tarkistus kummassa vaiheessa tahansa, vahva tyypitys takaa, että ohjelma ei voi tehdä kaikkein karkeimpia järjettömyyksiä, kuten laskea kahta listaa yhteen.

Vahvan staattisen tyypityksen tarkoitus on rajoittaa sallittujen, kääntyvien ohjelmien joukkoa niin, että mahdollisimman moni ohjelma, jossa ei ole mieltä, olisi tyyppivirheellinen. Jos tyyppijärjestelmä on kovin jäykkä, samalla menetetään kuitenkin suuri joukko mielekkäitä ohjelmia. Esimerkiksi käy Pascalin tyyppijärjestelmä, jossa jokaiselle tyypille on luotava oma listatyyppi ja kirjoitettava listaa käsittelevät funktiot. Ongelman kiertämiseksi on kehitetty rikkaampia polymorfisia tyyppijärjestelmiä, jotka sallivat samojen funktioiden käytön eri tyyppisten arvojen käsittelemiseen. Toinen staattisen tyypityksen ongelma on tyyppien esiin kirjoittamisesta ohjelmoijalle koituva työ ja ohjelmakoodin sotkeentuminen runsaista tyyppimäärityksistä. Tähän onglemaan auttavat tyypinpäättelyalgoritmit. Esimerkiksi ML:n ja Haskellin käyttämä Hindley-Milner-tyyppijärjestelmä kykenee automaattisesti päättelemään lähes kaikkien ohjelmoijan kirjoittamien funktioiden tyypit.

Vaikka kehittyneet tyyppijärjestelmät kykenevät elegantisti ilmaisemaan runsaan joukon ohjelmia, on olemassa paljon käytännön ongelmia, joihin niiden ilmaisuvoima ei riitä. Seuraavassa esitellään muutamia tällaisia ongelmia.

1.1 `Printf`-funktion tyyppi

C-kielen `printf`-funktio on kätevä apuneuvo tekstimuotoisen tulostuksen muotoiluun. Se on myös esimerkki funktiosta, jonka määritteleminen vahvasti tyypitetyssä kielessä on hankalaa [Aug98]. C-kielen `printf` saa argumenttinaan merkkijonon, johon on upotettu muotoilukomentoja, sekä muotoilumerkkijonosta riippuvan määrän muita argumentteja. Myös lisäargumenttien tyypit riippuvat muotoilumerkkijonosta. Koska C-ohjelmoija voi halutessaan ohittaa kielen tyyppijärjestelmän, voidaan argumenttien arvot tulkita muotoilumerkkijonon määräämiin tyyppeihin kuuluviksi vasta ohjelmaa ajettaessa. Jos argumentit vastaavat muotoilumerkkijonoa, funktio toimii hyvin. Muussa tapauksessa ohjelma saattaa tulostaa jotain kummallista tai kaatua epämääräiseen muistinosoitusvirheeseen. Rutiinia `printf` vastaavan funktion voisi kirjoittaa vahvasti tyypitetyssä Haskellissa esimerkiksi seuraavasti:

```
printf fmtstr = pr fmtstr "" where
   pr ""          res = res
   pr ('%':'d':cs) res = \i -> pr cs (res ++ show (i::Int))
   pr ('%':'s':cs) res = \s -> pr cs (res ++ s)
   pr ('%':c:cs)   res =       pr cs (res ++ [c])
   pr (c:cs)       res =       pr cs (res ++ [c])
```

Valitettavasti funktiolle ei voida antaa tyyppiä, sillä `pr`-funktion ensimmäisen haaran tyyppi on `String` → `(String` → `String)`, mutta toisen haaran tyyppi onkin `String` → `(String` → `Int` → `String)`. Toisin sanoen funktion tyyppi riippuu sen ensimmäisen argumentin arvosta, mitä Haskellin tyyppijärjestelmä ei pysty ilmaisemaan.

1.2 Ajoaikaiset virheet

Tyyppiturvallisen kielen tyyppijärjestelmän on estettävä järjettömyydet kuten taulukon indeksointi liian suurella arvolla tai tyhjän listan alkioiden käsittely. Vaikka kyseessä olisi staattinen tyypitys, on kääntäjän siksi lisättävä ohjelmaan tarkistus aina kun taulukon tai listan alkioita käsitellään. Ohjelman jokainen taulukon indeksointi tai listan ensimmäisen alkion käsittely on siis mahdollinen ajoaikaisen virheen esiintymispaikka. Ylimääräisten tarkistusten teko myös kuluttaa turhaan laskentaresursseja, ja nopeutta vaativat ohjelman osat on usein ohjelmoitava matalamman tason kielellä, kuten C:llä. Kuitenkin suurin osa tarkistuksista pystyttäisiin tekemään jo ohjelman käännösvaiheessa, jos tyyppijärjestelmä olisi tarpeeksi ilmaisuvoimainen [XiP98]. Tällöin havaittaisiin mahdolliset väärinkäytökset jo käännettäessä, mikä lisää ohjelman luotettavuutta, ja ylimääräisten tarkistusten puuttuminen nopeuttaisi ohjelmaa joissain tapauksissa huomattavasti.

Funktionaalisten kielten standardifunktio `zip` ottaa argumentteinaan kaksi listaa ja palauttaa listan, jossa on pareittain argumenttilistoissa vastaavilla paikoilla olevat alkiot. Haskellissa `zip` voidaan määritellä seuraavasti:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

Esimerkiksi "`zip [1,2] [3,4]`" palauttaa listan `[(1,3),(2,4)]`. `Zip` on hyvin määritelty vain, jos listat ovat yhtä pitkät. Jos funktiolle annetut listat ovat eri pituiset, on seurauksena ajoaikainen virhe [1]. Tämä voitaisiin estää, jos listan tyyppiin liitettäisiin tieto listan pituudesta [Xi99]. Jo käännösvaiheessa voitaisiin tarkistaa, että `zip`in saamat listat ovat yhtä pitkät. Paitsi että ajoaikaisen virheen mahdollisuus häviäisi, kääntäjä voisi tällöin generoida hieman tehokkaampaa koodia, kun joka askeleella riittäisi tarkistaa vain toisen listan tyhjyys. Oletetaan, että Haskelliin listatyyppiin lisätään listan pituuden ilmoittava määre, merkitään `[a](n)` n-alkioista listaa tyypin `a` arvoista. Tällöin `zip`-funktion tyyppi voidaan merkitä:

```
zip :: (n::Int) [a](n) -> [b](n) -> [(a,b)](n)
```

Listan pituuden lisääminen sen tyyppiin johtaa jälleen funktioihin, joiden tyyppi riippuu argumentin arvosta. Yksinkertaisena esimerkkinä `take`-funktio, joka saa argumentteina listan l ja luvun n. `Take` palauttaa listan, joka sisältää l:n n ensimmäistä alkiota. Jos listan pituus sisältyy sen tyyppiin, `take`n tyyppi riippuu selvästi argumentin n arvosta.

[1] Haskellin standardi-`zip` ei tuota virheilmoitusta, vaan lopettaa parituksen, kun jompi kumpi argumenttilistoista loppuu.

1.3 Järjestysrelaatio ja algebralliset ominaisuudet

Kehittyneissä polymorfisissa tyyppijärjestelmissä on mahdollista määritellä järjestämisalgoritmit niin, että ne voivat järjestää minkä tahansa tyypin alkioita, kunhan tyypille on määritelty kahden alkion järjestyksen määräävä funktio. Myös monet tietorakenteet, kuten keot ja hakupuut, toimivat ainoastaan tyypeillä, joilla on olemassa järjestysfunktio. Javassa voidaan järjestysfunktio määrittää rajapintaluokassa (interface), joka järjestettävän tyypin tulee toteuttaa. Haskellissa voidaan vastaavaan tarkoitukseen käyttää tyyppiluokkaa (type class). Tällaiset rajapinnat eivät kuitenkaan takaa, että järjestysfunktion määrittelemä relaatio todella on järjestysrelaatio, vaan se voi olla mikä tahansa totuusarvoinen funktio. Ainoa tapa yrittää varmistautua siitä, että tyyppijärjestelmälle liian vaativat rajoitukset tulevat täytetyiksi, on varustaa ohjelma sopivin kommentein ja reunahuomautuksin. Niinpä oikein ohjelmoitu järjestämisalgoritmi tai hakupuu saattaa toimia väärin, jos sen käsittelemän tyypin järjestysfunktiossa on virhe. Jos tyyppijärjestelmä kykenisi ilmaisemaan järjestysrelaation vaatimukset, havaittaisiin tällaisetkin virheet jo ohjelmaa käännettäessä.

Järjestysrelaation lisäksi olisi hyödyllistä antaa tyyppi esimerkiksi ekvivalenssirelaatioille ja erilaisille algebrallisille rakenteille ja ominaisuuksille. Mitä tarkemmin tyypin ominaisuudet määritellään, sitä paremmat mahdollisuudet kääntäjällä on havaita ohjelmointivirheet ja tuottaa tehokkaampaa koodia. Esimerkiksi punamustan hakupuun solmujen tyyppiin on tarpeeksi voimakkaalla tyyppijärjestelmällä mahdollista koodata puna-musta-ehto, jolloin tyypintarkistin voi käännösvaiheessa tarkistaa, että puuta käsittelevä algoritmi säilyttää puna-musta-ehdon [XiP99].

Esimerkkinä algebrallisen rakenteen esittämisestä tyyppinä on puoliryhmä. Puoliryhmä on joukko S varustettuna assosiatiivisella binaarioperaattorilla. Merkitään operaattoria symbolilla "+". Puoliryhmä voitaisiin ohjelmointikielessä ilmaista parina $(S, +)$, jossa S on joukko ja funktion $+$ tyyppi on $S \to S \to S$. Parin toisen komponentin tyyppi riippuu tässä tapauksessa ensimmäisestä komponentista. Lisäksi olisi esitettävä myös operaattorin $+$ assosiatiivisuus.

1.4 Riippuvat tyypit

Edellä käsitellyissä esimerkeissä ongelmaksi paljastui aina sama ilmiö: Funktion tai monikon tyyppi on oltava kiinteä; se ei saa riippua argumentin tai komponentin arvosta. Selkeästi moneen ongelmaan saataisiin ratkaisu, jos kieleen lisättäisiin *riippuvat tyypit*. Kuinka ilmaisuvoimainen tyyppijärjestelmä saadaan aikaan sallimalla funktion tyypin riippuminen sen argumenttien arvoista? Jos sallitaan mielivaltaiset riippuvuudet ja täysi rekursio tyyppimäärittelyissä, voidaan funktion tyyppiin koodata koko predikaattilogiikka. Tämä lähestymistapa on käytössä *konstruktiivisessa tyyppiteoriassa*, josta kerrotaan seuraavassa luvussa. Vielä ilmaisuvoimaisempaa tyyppijärjestelmää on vaikea kuvitella, mutta voimalla on kääntöpuolensa: täysimittaisten riippuvien tyyppien tyypintarkistus on ratkeamaton ongelma. Kompromissiratkaisuna voidaan sallia riippuvien tyyppien rajoitettu muoto, jolloin ei voida luoda kaikkia mahdollisa riippuvia tyyppejä, vaan jokin niiden mielekäs osajoukko. Tällöin saadaan joitain ongelmia ratkaistuksi, mutta tyypintarkistus säilyy ratkeavana. Esimerkiksi ML-kieleen on olemassa laajennus DML(C) [XiP99], jossa tyypit voidaan indeksoida totuusarvoilla ja kokonaislukujen lineaarisilla epäyhtälöillä sekä näistä loogisilla operaattoreilla yhdistellyillä kaavoilla. Tämä riittää esimerkiksi luvun 1.2 taulukon indeksointiin ja listojen pituuteen liittyviin ongelmiin.

2 Konstruktiivinen tyyppiteoria

2.1 Konstruktiivinen matematiikka

Matematiikassa käytetään usein ristiriitatodistusta: näytetään, että jos oletetaan todeksi todistettavan lauseen negaatio, seuraa jokin ristiriita. Tästä päätellään, että todistettavan lauseen on oltava tosi. *Konstruktiivisessa matematiikassa* ei tällainen epäsuora todistus kelpaa, vaan todistuksen on oltava todistettavan lauseen määrittelemä olio tai menetelmä, jolla tarvittavat ominaisuudet omaava olio voidaan konstuoida. Otetaan esimerkiksi lause: *On olemassa kaksi irrationaalikukua x ja y, joilla x^y on rationaalinen.* Tämä voidaan todistaa klassisen matematiikan keinoin seuraavasti: Luku $\sqrt{2}^{\sqrt{2}}$ on joko rationaalinen tai irrationaalinen. Jos se on rationaalinen, niin valitaan $x = \sqrt{2}$ ja $y = \sqrt{2}$. Jos se on irrationaalinen, valitaan $x = \sqrt{2}^{\sqrt{2}}$ ja $y = \sqrt{2}$, jolloin $x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$. Kummassakin tapauksessa siis halutut x ja y on olemassa, mutta todistus ei anna lukuja, jotka todistavat olemassaoloväitteen. Konstruktiivinen todistus antaisi ensin luvut x ja y, esimerkiksi $x = \sqrt{2}$ ja $y = 2 \cdot \log_2 3$, jotka ovat irrationaalisia, sekä lauseen todistuksen näillä luvuilla: $2^{\frac{2 \cdot \log_2 3}{2}} = 3$.

Konstruktiivisessa matematiikassa lause "on olemassa" tulkitaan "voidaan konstruoida" ja olemassaolo todistetaan näyttämällä eksplisiittisesti, miten haluttu olio voidaan tuottaa. Myös muiden loogisten lausekkeiden tulkinta poikkeaa klassisesta matematiikasta:

- Lauseen $p \vee q$ todistukseen tarvitaan joko p:n tai q:n todistus.
- Lauseen $p \wedge q$ todistukseen tarvitaan sekä p:n että q:n todistus.
- Lauseen $p \Rightarrow q$ todistus on algoritmi, joka annettuna p:n todistus tuottaa q:n todistuksen.
- Lauseen $\neg p$ todistus on algoritmi, joka annettuna p:n todistus tuottaa ristiriidan.

- Lauseen $\exists x P(x)$ todistukseen tarvitaan jokin olio x, jolla $P(x)$ on (konstruktiivisesti) tosi, sekä lauseen $P(x)$ todistus.
- Lauseen $\forall x P(x)$ todistus on algoritmi, joka tuottaa mille tahansa oliolle x lauseen $P(x)$ todistuksen.

Erityisesti klassisen logiikan kolmannen poissuljetun laki, jonka mukaan väite $P \vee \neg P$ on aina tosi, ei päde konstruktiivisessa matematiikassa. Konstruktiivinen matemaatikko uskoo väitteen $P \vee \neg P$ todeksi vasta, kun hänelle annetaan joko P:n tai $\neg P$:n todistus. Tämän takia ristiriitatodistus ei konstruktiivisessa matematiikassa kelpaa todistukseksi.

Tunnetuin konstruktiivisen matematiikan formalisointi lienee ruotsalaisen Per Martin-Löfin konstruktiivinen tyyppiteoria. Konstruktiivinen tyyppiteoria on eräänlainen metateoria, jota voidaan laajentaa kuvaamaan monenlaisia teorioita, esimerkiksi luonnollisten lukujen teoriaa tai naivia joukko-oppia. Nämä teoriat voidaan esittää konstruktiivisen tyyppiteorian sisällä määritelminä ja lauseina. Konstruktiivinen matematiikka ja sen formalisoinnit ovat erilaisten *automaattisten todistusjärjestelmien* pohjana.

2.2 Propositiot ovat tyyppejä

Konstruktiivinen tyyppiteoria voidaan käsittää myös puhtaana funktionaalisena ohjelmointikielenä tulkitsemalla sen propositiot ohjelmointikielen tyypeiksi. Tämä tyyppien ja propositioiden välinen yhteys tunnetaan nimellä *Curry-Howard-isomorfismi*. Kaava $p \in P$ luetaan yleensä ohjelmointikielten yhteydessä: "Olio p kuuluu joukkoon P" tai "p:n tyyppi on P." Toisaalta P voidaan tulkita väitteeksi ja p sen konstruktiiviseksi todistukseksi, jolloin kaava $p \in P$ voidaankin lukea: "p on väitteen P todistus." Epätodet väitteet, joilla ei ole yhtään todistusta, samastuvat tällöin tyhjään joukkoon $\emptyset$. Esimerkiksi lause $0 \in \mathbb{N}$ voitaisiin lukea toisaalta: "0 on luonnollinen luku" tai "luonnollisia lukuja on olemassa, sillä 0 on luonnollinen luku". Taulukkoon 1 on koottu loogiset konnektiivit ja niiden tulkinnat tyyppeinä. Konstruktiivisesta tyyppiteoriasta on olemassa suuri joukko erilaisia versioita, ja eri tahot käyttävät eri merkintöjä. Tässä esitelty versio ja merkinnät ovat sekoitus useiden eri lähteiden kuvauksista [Bac90, Coq94, NPS94].

Riippuva tulotyyppi $\sum(P, x.Q(x))$ voidaan käsittää myös alijoukkotyyppinä. Tyyppiin kuuluvat kaikki tyypin P alkiot, jotka toteuttavat ehdon Q. Alkiot on myös varustettu todistuksella Q:n toteutumisesta.

Tautologisilla propositioilla on tyyppeinä tulkittuna mielenkiintoinen ominaisuus: tautologiaa vastaava tyyppi ei koskaan ole tyhjä, vaan aina voidaan ilman oletuksia luoda tautologiatyypin alkio. Esimerkiksi funktiotyyppi $A \to A$ on tautologia, sillä identtinen kuvaus $\lambda(x \in A).x$ kuuluu siihen. Sen sijaan kuvausta $A \to B$ vastaavaa funktiota ei voida ilman lisäoletuksia konstruoida. Ei ole mitään tapaa kirjoittaa funktiota, joka ottaa mielivaltaisen tyypin A jäsenen ja palauttaa mielivaltaisen tyypin B jäsenen.

Propositio	Tyyppi	Tyypin tulkinta
$P \Rightarrow Q$	$P \to Q$	Funktio tyypiltä P tyypille Q
$P \wedge Q$	$P \times Q$	Tyyppien karteesinen tulo; pari tai tietue, joka koostuu sekä tyypin P että tyypin Q alkiosta.
$P \vee Q$	$P + Q$	Summatyyppi; "tagged type", joka koostuu joko P:n tai Q:n alkiosta sekä "tagista", joka ilmoittaa kumpaa tyyppiä alkio on kyseessä.
$\exists(P, x.Q(x))$	$\sum(P, x.Q(x))$	Riippuva tulotyyppi; koostuu pareista (x, y), missä $x \in P$ ja $y \in Q(x)$.
$\forall(P, x.Q(x))$	$\prod(P, x.Q(x))$	Riippuvien funktioiden tyyppi; koostuu funktioista, jotka tuottavat mille tahansa $x \in P$ olion $y \in Q(x)$. y:n tyyppi voi riippua x:stä.
$\neg P$	$P \to \emptyset$	

Taulukko 1: Propositioiden ja tyyppien vastaavuus

2.3 Esimerkki ohjelmoinnista tyyppiteoriassa

Konstruktiivisen tyyppiteorian käyttöä ohjelmointikielenä on parasta tarkastella esimerkin kautta [Coq94, Mäe96]. Ohjelman kirjoittaminen vastaa Curry-Howard-isomorfismin mukaisesti todistuksen etsimistä ohjelman tyyppiä vastaavalle väitteelle. Esimerkkiohjelma on funktio, joka etsii annetun kokonaislukulistan pienimmän alkion. Ohjelma on konstruktiivinen todistus väitteelle:

> *Olkoon L epätyhjä konaislukulista. On olemassa luku n, joka on pienin listan L alkio.*

Konstruktiivinen todistus väitteelle on algoritmi, joka annettuna lista L tuottaa sen pienimmän alkion. On siis rakennettava tyyppi, joka vastaa väitettä ja etsittävä ohjelma, joka on tätä tyyppiä. Väite kirjoitettuna edellisen luvun notaatiolla on:

$$(ns \in NSeq) \to \exists(\mathbb{N}, n.\mathrm{Minimal}(n, ns)).$$

Tässä $NSeq$ on epätyhjien lukulistojen tyyppi. Predikaatti $\mathrm{Minimal}(n, ns)$ kertoo, että n on pienin listan $NSeq$ luvuista. Jotta lause olisi mielekäs, on ensin annettava määritelmät luonnollisille luvuille, niiden listoille sekä Minimalpredikaatille. Määritellään ensin luonnolliset luvut joukkona ja konstruktoreilla, jotka tuottavat joukon alkiot:

$$
\begin{aligned}
&\mathbb{N} \in Set \\
&\quad 0 \in \mathbb{N} \\
&\quad s \in \mathbb{N} \rightarrow \mathbb{N}
\end{aligned}
$$

Määritelmän ensimmäinen rivi kertoo, että luonnolliset luvut $\mathbb{N}$ on joukko. Toinen rivi kertoo, että 0 on luonnollinen luku. Kolmannella rivillä kerrotaan, että seuraajafunktio s tuottaa luonnollisen luvun. Se voidaan lukea: "Olkoon n luonnollinen luku. Tällöin myös $s(n)$ on luonnollinen luku." Seuraavaksi on määriteltävä lista. On mahdollista määritellä geneerinen lista, joka voi sisältää minkä tahansa tyyppisiä alkioita, mutta esimerkin yksinkertaistamiseksi määritellään luonnollisten lukujen lista:

$$
\begin{aligned}
&NSeq \in Set \\
&\quad single \in \mathbb{N} \rightarrow NSeq \\
&\quad cons \in (\mathbb{N}; NSeq) \rightarrow NSeq
\end{aligned}
$$

Listan konstruktorit ovat *single*, joka ottaa luvun ja palauttaa yksialkioisen listan, sekä *cons*, joka ottaa uuden listan päähän laitettavan luvun sekä listan hännän. Määritelmät ovat tuttuja perinteisistä funktionaalisista kielistä. Konstruktorit voidaan lukea myös väitteinä: "Olkoon n luonnollinen luku. Tällöin $single(n)$ on lista. Olkoon n luonnollinen luku ja s lista. Tällöin $cons(n, s)$ on lista."

Jäljellä on predikaatin Minimal määrittely. Tavanomaisessa ohjelmoinnissa tällaiset predikaatit määritellään yleensä totuusarvoisena funktiona. Konstruktiivisessa tyyppiteoriassa predikaatti määritellään joukkona ja sen konstruktoreina, aivan kuten listat ja luonnolliset luvut määriteltiin. Erona on, että Minimal-predikaatin alkiot riippuvat annetusta luvusta ja listasta; se on itse asiassa $\mathbb{N}$:n ja $NSeq$:n yli indeksoitu joukkoperhe, jonka alkiot ovat todistuksia lauseelle: "Luku n on listan s pienin alkio." Jos n ei ole s:n pienin alkio, ei lauseelle ole todistusta, joten esimerkiksi joukko $\mathrm{Minimal}(0, single(s(0)))$ on tyhjä joukko. Seuraavassa esitellään predikaatti Minimal ja sen konstruktorit:

$$
\begin{aligned}
&\mathrm{Minimal} \in (n \in \mathbb{N}; s \in NSeq) \rightarrow Set \\
&\ \mathrm{m1} \quad \in \quad (n \in \mathbb{N}) \rightarrow \mathrm{Minimal}(n, single(n)) \\
&\ \mathrm{m2} \quad \in \quad ((n \in \mathbb{N}); (m \in \mathbb{N}); (s \in NSeq); \mathrm{Minimal}(n, s); \mathrm{Leq}(m, n)) \\
&\qquad\qquad \rightarrow \mathrm{Minimal}(m, cons(m, s)) \\
&\ \mathrm{m3} \quad \in \quad ((n \in \mathbb{N}); (m \in \mathbb{N}); (s \in NSeq); \mathrm{Minimal}(n, s); \mathrm{Leq}(n, m)) \\
&\qquad\qquad \rightarrow \mathrm{Minimal}(n, cons(m, s))
\end{aligned}
$$

Konstruktori m1 ilmoittaa, että tyhjän listan ainoa alkio on sen pienin alkio. Konstruktori m2 määrittelee tapauksen, jossa *cons*-konstruktorilla muodostetun listan pää on suurempi tai yhtä suuri kuin hännän pienin alkio. M3 määrittelee tapauksen, jossa listan pää on pienempi tai yhtä suuri kuin hännän pienin alkio. Esimerkiksi m2 voidaan ajatella funktiona, joka ottaa kaksi lukua n ja m, listan s, todistuksen, että n on listan pienin alkio sekä todistuksen, että m on pienempi tai yhtä suuri kuin n. Funktion arvo on todistus väitteelle: "Luku m on listan $cons(m, s)$ pienin luku." Määritelmä ei valitettavasti ole vieläkään täydellinen: Minimal-predikaatin määritelmä sisältää uuden predikaatin Leq, joka määritellään seuraavaksi:

$$
\begin{aligned}
&\mathrm{Leq} \in (n \in \mathbb{N}; m \in \mathbb{N}) \to Set\\
&\quad \mathrm{leq}_0 \in (n \in \mathbb{N}) \to \mathrm{Leq}(0, n)\\
&\quad \mathrm{leq}_s \in (n \in \mathbb{N}; m \in \mathbb{N}; \mathrm{Leq}(m, n)) \to \mathrm{Leq}(s(m), s(n))
\end{aligned}
$$

Leq-predikaatti ei kuitenkaan ole vielä valmis käytettäväksi. Järjestysrelaation tulee olla transitiivinen ja täydellinen. Täydellisyys tarkoittaa, että kaikilla m ja n on oltava joko Leq(m, n) tai Leq(n, m). Listan pienimmän alkion etsivän ohjelman kannalta transitiivisuus ei kuitenkaan ole oleellista, joten se jätetään tässä todistamatta. Sen sijaan täydellisyyden todistusta tarvitaan myöhemmin. Todistus Leq:n täydellisyydelle on tyypin $(n \in \mathbb{N}; m \in \mathbb{N}) \to$Or(Leq$(m, n)$,Leq$(n, m)$) alkio. Looginen tai-operaatio Or ja sen tyyppinen Leq:n täydellisyyden todistava alkio leq_total määritellään seuraavasti:

$$
\begin{aligned}
&\mathrm{Or} \in (A \in Set; B \in Set) \to Set\\
&\quad \mathrm{or}_1 \in ([A \in Set; B \in Set]; a \in A) \to \mathrm{Or}(A, B)\\
&\quad \mathrm{or}_2 \in ([A \in Set; B \in Set]; b \in B) \to \mathrm{Or}(A, B)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{leq\_total} \in (n \in \mathbb{N}; m \in \mathbb{N}) \to \mathrm{Or}(\mathrm{Leq}(n, m),\mathrm{Leq}(m, n))\\
&\quad \mathrm{leq\_total}\ (0, n) \equiv \mathrm{or}_1(\mathrm{leq}_0(n))\\
&\quad \mathrm{leq\_total}\ (s(m), 0) \equiv \mathrm{or}_2(\mathrm{leq}_0(s(m)))\\
&\quad \mathrm{leq\_total}\ (s(m), s(n)) \equiv\\
&\qquad \mathrm{case\ leq\_total}(m, n)\ \mathrm{of}\\
&\qquad\quad \mathrm{or}_1(l) \Rightarrow \mathrm{or}_1(\mathrm{leq}_s(s(n), s(m), l))\\
&\qquad\quad \mathrm{or}_2(l) \Rightarrow \mathrm{or}_2(\mathrm{leq}_s(s(m), s(n), l))\\
&\qquad \mathrm{end}
\end{aligned}
$$

Or-predikaatin määrittelyssä uusi piirre ovat hakasulkeisiin merkityt argumentit, joita ei tarvitse eksplisiittisesti merkitä. Leq_total on oikeastaan funktio, joka annettuna kaksi luonnollista lukua n ja m palauttaa todistuksen joko väitteelle Leq(n, m) tai väitteelle Leq(m, n). Määrittelyn ensimmäinen rivi kertoo leq_total:in tyypin. Sen jälkeen funktion arvo määritellään hahmoyhtälöillä kaikilla mahdollisilla m:n ja n:n konstruktoriyhdistelmillä.

Nyt halutun ohjelman tyyppi on vihdoin kokonaan määritelty, joten ohjelma voidaan kirjoittaa antamalla todistus sen tyyppiä vastaavalle väitteelle. Tämä tehdään luomalla ohjelman tyyppiin kuuluva alkio min:

$$
\begin{array}{l}
\min \in (ns \in NSeq) \to \exists(\mathbb{N}, n.\mathrm{Minimal}(n, ns)) \\
\quad \min(single(n)) \equiv \exists\text{-}intro(n, \mathrm{m1}(n)) \\
\quad \min(cons(n, s)) \equiv \\
\quad\quad \text{case } \min(s) \text{ of} \\
\quad\quad\quad \exists\text{-}intro(m, m_0) \Rightarrow \\
\quad\quad\quad\quad \text{case leq\_total}(m, n) \text{ of} \\
\quad\quad\quad\quad\quad \mathrm{or}_1(l) \Rightarrow \exists\text{-}intro(m, \mathrm{m2}(n, m, s, m_0, l) \\
\quad\quad\quad\quad\quad \mathrm{or}_2(l) \Rightarrow \exists\text{-}intro(n, \mathrm{m1}(n, m, s, m_0, l) \\
\quad\quad\quad\quad \text{end} \\
\quad\quad \text{end}
\end{array}
$$

Lopullinen ohjelma muistuttaa tavanomaisella funktionaalisella kielellä kirjoitettua ohjelmaa. Ensimmäinen hahmoyhtälö määrittelee funktion tapauksessa, jossa argumenttilista on yksialkioinen. Tällöin väite "on olemassa luku n, joka on annetun listan pienin alkio" voidaan todistaa muodostamalla pari, jonka ensimmäinen komponentti on listan ainoa alkio ja toiden komponentti todistus väitteelle, jonka mukaan tämä alkio on listan pienin. Todistukseksi kelpaa Minimal-predikaatin ensimmäinen konstruktori, joka kertoo, että single-konstruktorilla muodostetun listan ainoa alkio on myös sen pienin. Merkintä $\exists$-$intro(a, b)$ tarkoittaa, että muodostetaan pari, jonka ensimmäinen komponentti on alkio, joka toteuttaa toisen komponentin edustaman väitteen.

Toinen hahmoyhtälö määrittelee cons-konstruktorilla muodostetun listan pienimmän alkion. Määritelmä viittaa rekursiivisesti itseensä. Suomeksi luettuna määritelmä on suurin piirtein: "Olkoon m listan s pienin alkio ja m_0 todistus lauseelle: 'm on listan s pienin alkio.' Leq_total(m, n) on todistus lauseelle 'm on pienempi tai yhtä suuri kuin n tai n on pienempi tai yhtä suuri kuin m.' Jos Leq_total(m, n) on todistettu näyttämällä, että $m \leq n$, voidaan lause 'on olemassa luku n_0, joka on listan $cons(n, s)$ pienin alkio' todistaa asettamalla $n_0 = m$ ja todistamalla se käyttämällä Minimal-predikaatin konstruktoria m2. Jos taas Leq_total(m, n) on todistettu näyttämällä, että $n \leq m$, voidaan lause 'on olemassa luku n_0, joka on listan $cons(n, s)$ pienin alkio' todistaa asettamalla $n_0 = n$ ja todistamalla se käyttämällä Minimal-predikaatin konstruktoria m1."

Tavanomaisessa ohjelmointikielessä min-ohjelman tyypiksi olisi annettu kuvaus listoilta luonnollisille luvuille. Tällaisen kielen kääntäjä pystyy ainoastaan takaamaan, että funktio todella palauttaa luvun sekä valvomaan, ettei sille yritetä syöttää muuta kuin listoja. Konstruktiivisen tyyppiteorian tyyppijärjestelmä sen sijaan kykenee ilmaisemaan täsmällisesti ohjelman tarkoituksen tyyppinä. Koska tyyppijärjestelmällä on propositiologiikan ilmaisuvoima, pystytään mielivaltainen ohjelman spesifikaatio koodaamaan tyypiksi ja automaattisesti tarkastamaan, että kirjoitettu ohjelma vastaa spesifikaatiota.

3 Riippuvia tyyppejä tukeva ohjelmointikieli

Yksikään yleisessä käytössä oleva ohjelmointikieli ei sisällä riippuvia tyyppejä edes rajoitetussa muodossa. Konstruktiiviseen tyyppiteoriaan tai muihin konstruktiivisen matematiikan formalisointeihin perustuvissa automaattisissa todistusjärjestelmissä ja todistuseditoreissa riipuvia tyyppejä sen sijaan on käytetty. *Cayenne* [Aug98, AuM98] on Haskellin tyylinen ohjelmointikieli, jonka tyyppijärjestelmä perustuu konstruktiiviseen tyyppiteoriaan. Cayennessa voi siis tyyppejä käyttää täydellisten spesifikaatioiden kirjoittamiseen tyypintarkistimen huolehtiessa niiden tarkistuksesta. Voidaan siis sanoa, että ohjelma, jonka Cayenne suostuu kääntämään, toimii oikein – ainakin, jos spesifikaatiossa ei ole virheitä. Valitettavasti vahvalla tyyppijärjestelmällä on rajoituksensa: koska Cayenne sallii täyden rekursion tyyppimäärittelyissä, vaatisi tyypintarkistuksen ratkaiseminen pysähtymisongelman ratkaisemista. Tämä kuulostaa pahalta. Tyypintarkistin saattaa joutua ikuiseen silmukkaan. Cayennen tekijän Lennart Augustssonin mukaan tämä kuitenkin on käytännössä harvoin ongelma. Jos tyypintarkistaja jää silmukkaan, on käännettävässä ohjelmassa yleensä virhe. Toisin kuin Haskellissa, Cayenne ei osa päätellä tyyppejä, vaan ne on aina kirjoitettava esiin. Riippuvien tyyppien monimutkaisuus tekee monessa tapauksessa Hindley-Milner-tyylisen tyypinpäättelyn mahdottomaksi.

Cayennen riippuvien tyyppien avulla voidaan printf-funktio määritellä tyyppiturvallisesti. Seuraavassa on luvun 1.1 Haskell-kielistä printf-funktiota vastaava Cayennella kirjoitettu printf. Siinä printf-funktion tyyppi on itsessään funktio, jonka laskeminen muistuttaa muodoltaan itse funktion laskemista.

```
PrintfType :: String -> #
   PrintfType ""           = String
   PrintfType ('%':'d':cs) = Int    -> PrintfType cs
   PrintfType ('%':'s':cs) = String -> PrintfType cs
   PrintfType ('%': _ :cs) =           PrintfType cs
   PrintfType ( _ :cs)     =           PrintfType cs

printf :: (fmt:String) -> PrintfType fmt
   printf fmt = pr fmt ""

pr :: (fmt::String) -> String -> PrintfType fmt
   pr ""           res = res
   pr ('%':'d':cs) res = \(i::Int)    -> pr cs (res ++ show i)
   pr ('%':'s':cs) res = \(s::String) -> pr cs (res ++ s)
   pr ('%':c:cs)   res =                 pr cs (res ++ [c])
   pr (c:cs)       res =                 pr cs (res ++ [c])
```

Cayennen tyypit ovat ensimmäisen luokan arvoja kuten funktiotkin. Niitä voidaan välittää funktion parametrina tai asettaa tietueiden kentiksi. Tästä syystä

Cayenne ei tarvitse erillistä moduulijärjestelmää, vaan tavallisia tietueita – joita konstruktiivisessa tyyppiteoriassa kutsutaan summatyypeiksi – voidaan käyttää moduuleina. Cayenne on kokeellinen kieli; kukaan ei ole soveltanut sitä tuotantokäyttöön. Kaikkine puutteineenkin Cayenne kuitenkin osoittaa, että riippuvat tyypit on mahdollista toteuttaa myös ohjelmointikielessä. Se osoittaa myös, että on vielä pitkä matka ohjelmointikieleen, jossa riippuvat tyypit on toteutettu niin, että niiden käytöstä on enemmän hyötyä kuin hankaluuksia.

Viitteet

[Aug98] Augustsson, L., Cayenne – a language with dependent types, 1998. `http://www.cs.chalmers.se/~augustss/cayenne/paper.ps` [9.3.2000].

[AuM98] Augustsson, L., Carlsson, M., An exercise in dependent types: a well typed interpreter, 1998. `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps` [9.3.2000].

[Bac90] Backhouse, R. C., Constructive type theory; A perspective from computing science. Dijkstra, E. W. (toim.), *Formal Developement of Programs and Proofs*, Addison-Wesley, 1990.

[Coq94] Coquand, T., Nordström, B., Smith, J. M., von Sydov, B., Type theory and programming. *Bulletin of the EATCS* 54, 1994.

[Mäe96] Mäenpää, P., Ohjelmointi konstruktiivisessa tyyppiteoriassa. Teoksessa Gefwert, C., Orponen, P., Seppänen, J. (toim.), *Logic, Mathematics and the Computer. Foundations: History, Philosophy and Applications*, Finnish Artificial Intelligence Society, Helsinki, 1996, 105–118.

[NPS94] Nordström, B., Petersson, K., Smith, J. M., Martin-Löf's type theory, 1994. `http://www.cs.chalmers.se/~bengt/papers/hlcs.ps.gz`. [9.3.2000]

[Xi99] Xi, H., Dead code elimination through dependent types. *Proc. First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*, Springer-Verlag LNCS vol. 1551, San Antonio, January 1999, 228–242.

[XiP98] Xi, H., Pfenning, F., Eliminating array bound checking through dependent types. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, Montreal, June 1998, 249–257.

[XiP99] Xi, H., Pfenning, F., Dependent types in practical programming (extended abstract). *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, San Antonio, January 1999 , 214–227.