Tyyppikvanttorit

Kari Laasonen

Helsinki 21.3.2003

Tyyppiteoria ja ohjelmointikielet -seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Tyyppikvanttoreita tarvitaan määriteltäessä tyyppijärjestelmiä, jotka sallivat (parametrisen) polymorfismin eli monimuotoisuuden. Lisättäessä monimuotoisuus yksinkertaiseen lambdakalkyyliin saadaan tärkeä kalkyyli, *järjestelmä F*. Tämä järjestelmä muodostaa perustan, jonka päälle rakennetaan myöhemmin kehittyneempiä tyyppijärjestelmiä.

Tämä kirjoitelma, joka perustuu B. Piercen oppikirjaan [Pie02, luvut 23 ja 24], käsittelee tyyppikvanttoreita ja niiden käyttöä järjestelmä F:ssä. Aluksi käydään läpi monimuotoisuuden lajeja. Luvussa 3 määritellään järjestelmä F ja tutustutaan monimuotoiseen lambdakalkyyliin esimerkein. Järjestelmä F:n käyttämän universaalikvanttorin ($\forall$) lisäksi myös eksistentiaalikvanttori ($\exists$) on käyttökelpoinen tyyppijärjestelmissä. Sen käyttöä datan abstrahointiin tarkastellaan luvussa 4.

2 Monimuotoisuus

Tyyppijärjestelmiä, jotka mahdollistavat saman ohjelmakoodin käytön usealla eri tyypillä, sanotaan *monimuotoisiksi* tai *polymorfisiksi*. Monimuotoisuus voidaan jakaa seuraaviin alalajeihin [CaW85]:

- *Parametrinen monimuotoisuus*, jota tämä kirjoitelma käsittelee, mahdollistaa ohjelmakoodin "geneerisen" käytön. Tämä tapahtuu käyttämällä tyyppimuuttujia. Tyyppimuuttujalle voidaan antaa arvo, joka on tyyppi ("instantiointi"), jolloin kirjoitetusta koodista saadaan aina uusi, arvotyypille sopiva versio. Itse koodi pysyy siis samana, vaikka tyyppi vaihtuisikin.

  Parametrista monimuotoisuutta on edelleen kahta lajia, mutta tähän palataan luvussa 3.4.

- *Sisällytysmonimuotoisuus* (inclusion polymorphism) käyttää alityypitystä. Jokaisella alityypillä voi olla oma versionsa monimuotoisesta aliohjelmasta. Näistä valitaan se, mikä tarkimmin vastaa olion tyyppiä.

- *Satunnainen monimuotoisuus* tarkoittaa, että monimuotoinen aliohjelma toteuttaa eri tyypeille jollain tapaa samanlaisen operaation. Toteuttava koodi voi olla eri tyypeille aivan erilainen, eikä tyypeilläkään tarvitse olla mitään erityistä yhteyttä.

  Kuormituksessa (overloading) usealla aliohjelmalla voi olla sama nimi:

  ```
  void print(int x);
  void print(String x);
  ```

  Näistä valitaan oikea parametrien määrän ja tyyppien perusteella.

  Toinen tapaus on *automaattinen tyypinmuunnos* (type coercion), joka tarkoittaa muunnosta, joka tehdään ilman, että ohjelmoija sitä pyytää. Esimerkiksi C-kielessä aritmetiikkaoperaattorit tekevät tällaisia muunnoksia.

Funktionaalisessa ohjelmoinnissa termillä "monimuotoisuus" tarkoitetaan lähes aina parametrista monimuotoisuutta. Olio-ohjelmoinnissa sillä tarkoitetaan yleensä taas sisällytysmonimuotoisuutta. Parametrisesta monimuotoisuudesta puhutaan *geneerisyytenä*. Tällaisia ovat esimerkiksi Adan `generic`- ja C++:n `template`-rakenteet.

Jos tyyppi-informaatiota pystytään käyttämään suoritusaikaisesti, satunnaisen monimuotoisuuden ilmaisuvoima kasvaa edelleen. Tähän lajiin kuuluvat tyypin kysely (Javan `instanceof`-operaattori) ja peilausmekanismit (reflection).

3 Järjestelmä F

Parametrinen monimuotoisuus voidaan rakentaa yksinkertaisesti tyypitetyn lambdakalkyylin päälle, aivan kuten on jo tehty alityyppien ja rekursiivisten tyyppien kanssa. Tämä laajennus, jonka ensimmäisenä suoritti Jean-Yves Girard vuonna 1972, on osoittautunut erittäin tärkeäksi, ja se onkin saanut oman nimen: *järjestelmä F* (system F). Myös termi *polymorfinen lambdakalkyyli* on käytössä.

3.1 Määritelmä

Yksinkertaisessa lambdakalkyylissä peruskäsite on tietysti (tyypitetty) lambda-abstraktio $\lambda\mathtt{x}:\mathtt{T}.\mathtt{t}$ ja sen soveltaminen ("`t t`"). Entä jos myös *tyyppi* ajateltaisiin muuttujaksi? Tyyppimuuttujille voidaankin määritellä vastaavat käsitteet:

- Tyyppiabstraktio $\lambda\mathtt{X}.\mathtt{t}$, jossa `X` on tyyppiparametri;[1]
- Tyypinsovellus (tai *instantiointi*) `t [T]`.

On melko selvää, kuinka tällaisia rakenteita käytetään. Tyypinsovellus antaa tyyppiabstraktiolle (konkreettisen) tyypin `T`, joka korvaa abstraktiossa käytetyn muuttujan `X`. Saman asian sanoo evaluointisääntö

$$(\lambda\mathtt{X}.\mathtt{t})\,[\mathtt{T}] \to [\mathtt{X} \mapsto \mathtt{T}]\mathtt{t} \qquad \text{(E-TAPPTABS)}$$

Tätä voi verrata tavalliseen lambda-abstraktion sovellukseen:

$$(\lambda\mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_{12})\,\mathtt{v}_2 \to [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-APPABS)}$$

Määritellään monimuotoinen identiteettifunktio

```
id = λX. λx:X. x;                                    (1)
```

Sovelluksella `id[T]` saadaan tyypin `T` identiteettifunktio, eli $\mathtt{id[T]\ t} \to \mathtt{t}$, kunhan vain `t` on tyyppiä `T`.

[1] Tästä käytetään joskus merkintää $\Lambda\mathtt{x}.\mathtt{t}$, mutta yhtä hyvin voidaan sopia, että kaikki isot kirjaimet ovat tyyppiparametreja.

Vielä täytyisi selvittää, mikä on funktion `id` tyyppi. Sovellus `id[T]` on selvästi funktio, joten se on tyyppiä $\mathtt{T}\to\mathtt{T}$ kaikilla parametrityypin `T` arvoilla. Jotta tämä saataisiin aikaan, täytyy funktiolla `id` siis olla "geneerinen" tyyppi, josta voidaan konstruoida haluttu tulostyyppi. Määritellään *universaalityyppi* $\forall\mathtt{X}.\mathtt{T}_1$, jossa $\mathtt{T}_1$ riippuu tyypistä `X`. Funktion `id` tyyppi on siis $\forall\mathtt{X}.\mathtt{X}\to\mathtt{X}$. Universaalityypeille pätevät seuraavat tyypityssäännöt:

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{X}.\mathtt{t}_2 : \forall\mathtt{X}.\mathtt{T}_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall\mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\,[\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \qquad \text{(T-TApp)}$$

Edellinen sääntö määrittelee tyyppiabstraktion tyypiksi universaalityypin, jälkimmäinen taas kertoo, mikä on tyyppisovelluksen tyyppi. Tyyppimuuttuja `X` on sisällytetty kontekstiin Γ. Tässä noudatetaan aiempaa konventiota, jonka mukaan kaikkien muuttujien (myös tyyppimuuttujien) nimien täytyy erota nimistä, jotka on jo määritelty kontekstissa Γ.

Monimuotoisuuden lisäksi järjestelmä F:ään voidaan lukea mukaan monet jo aiemmin käsitellyt ominaisuudet. Tällaisia ovat esimerkiksi tietueet, luonnolliset luvut ja totuusarvot, sekä termit `let` ja `fix`.

3.2 Esimerkkejä

Tarkastellaan muutamia yksinkertaisia esimerkkejä. Määritellään kahdennusfunktio

```
  double = λX. λf:X→X. λa:X. f (f a);
▸ double : ∀X. (X→X) → X→X
```

Funktion `double` tyyppi seuraa suoraan tyypityssäännöistä. Sisimmän abstraktion tyyppi on säännön T-Abs ja kontekstin mukaan $\mathtt{X}\to\mathtt{X}$. Keskimmäisen abstraktion tyyppi on samoin $\mathtt{Y}\to\mathtt{Y}$, missä `Y` on $\mathtt{X}\to\mathtt{X}$. Uloimman tyyppiabstraktion tyyppi saadaan suoraan säännöstä T-TAbs. Funktiota käytetään soveltamalla sitä johonkin tyyppiin, jonka jälkeen annetaan sopivasti tyypitetyt parametrit `f` ja `a`:

```
  double [Nat] (λx:Nat. succ(succ(x))) 3
▸ 7 : Nat
```

Parametrin soveltaminen itseensä ($\lambda\mathtt{x}.\mathtt{x}\ \mathtt{x}$) ei tyypity yksinkertaisessa lambdakalkyylissä (ilman rekursiivisiä tyyppejä). Järjestelmä F:ssä voidaan kuitenkin sanoa

```
  self-appl = λx:∀X.X→X. x [∀X.X→X] x;
▸ self-appl : (∀X. X→X) → (∀X. X→X)
```

Ensimmäinen `x` instantioidaan universaalityypillä, jolloin siitä tulee kuvaus universaalilta funktiolta toiselle. (Tähän ominaisuuteen palataan luvussa 3.4.) Monimuotoisen `id`:n lisäksi on vaikea keksiä muita funktioita, joita tässä yhteydessä voisi käyttää.

Tyypittämättömässä lambdakalkyylissä oli mahdollista määritellä totuusarvot ja numeraalit funktioina, *Church-koodauksina*. Tällainen koodaus on mahdollista suorittaa myös järjestelmä F:ssä ja siten saada koodaukset tyypitettyä. Koska tämä Church-koodaus on mahdollista tehdä, ei koodattujen arvojen lisääminen myöhemmin primitiiveiksi muuta järjestelmän laskentaominaisuuksia. Järjestelmä F:n ilmaisuvoima onkin huomattava.

Tarkastellaan nyt Churchin koodausta totuusarvoille. Tyypittämättömässä kalkyylissä oli

```
tru = λt. λf. t;
fls = λt. λf. f;
```

Tavoitteena on selvittää tyyppi, joka voidaan antaa näille molemmille. Kummankin parametrin `t` ja `f` tulee selvästi olla samaa (vapaavalintaista) tyyppiä. Tästä seuraa, että `tru` ja `fls` ovat tyyppiä `∀X.X→X→X`, jota voidaan merkitä symbolilla `CBool`. Saadaan siis

```
tru = λX. λt:X. λf:X. t;
fls = λX. λt:X. λf:X. f;
```

Esimerkkinä totuusarvojen käytöstä on komplementtifunktio

```
not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
```

joka on kuvaus yhdeltä universaalityypiltä toiselle. Totuusarvo `b` on monimuotoinen, joten se täytyy instantioida tyypillä `X` ennen sovellusta.

Vastaavaan tapaan voidaan määritellä myös tyypitetyt Churchin numeraalit. Totuusarvojen tapaan numeraalien tyyppi on pääteltävissä suoraan niiden funktiomuotoisesta esityksestä. Määrittely ja esimerkkejä löytyy Piercen kirjasta, §23.4.

3.3 Monimuotoinen listarakenne

Klassinen esimerkki monimuotoisuudesta on tietorakenne, esimerkiksi lista, johon voidaan tallettaa haluttua tyyppiä olevia alkioita. Jos käytetty ohjelmointikieli tukee monimuotoisuutta, lista voidaan ohjelmoida vain kerran ja instantioida sen jälkeen useille eri tyypeille.

Ryhdytään nyt kirjoittamaan listarakennetta järjestelmä F:lle. Oletetaan ensin, että seuraavat listaprimitiivit ovat jo olemassa:

```
nil : ∀X. List X                    head : ∀X. List X → X
isnil : ∀X. List X → Bool           tail : ∀X. List X → List X
cons : ∀X. X → List X → List X
```

Näiden avulla voidaan toteuttaa erilaisia monimuotoisia listaoperaatioita. Esimerkkinä olkoon `map`. Sen parametreinä on lista, jonka alkiot ovat tyyppiä `X`, ja funktio `f`: `X` → `Y`, ja se palauttaa listan, jonka alkiot ovat tyyppiä `Y`:

```
map = λX. λY.
       λf: X→Y.
         (fix (λm: (List X)→(List Y). λl: List X.
                if isnil [X] l
                  then nil [Y]
                  else cons [Y] (f (head [X] l))
                                (m (tail [X] l))));
```

▸ `map : ∀X. ∀Y. (X→Y) → List X → List Y`

```
l = cons [Nat] 3 (cons [Nat] 4 (nil [Nat]));
```

▸ `l : List Nat`

```
head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
```

▸ `5 : Nat`

Mainitut listaprimitiivitkin voidaan toteuttaa puhtaassa järjestelmä F:ssä käyttäen Churchin koodausta. Ideana on esittää lista sisäkkäisinä funktiokutsuina, kuten OCaml:ssa voidaan tehdä:

$$\texttt{List.fold\_right c [x;y;z] n} \equiv \texttt{c x (c y (c z n))} \tag{2}$$

Tämän mukaisesti määritellään listatyyppi

```
List X = ∀R. (X→R→R) → R→R;
```

Listan elementit ovat tyyppiä `X`, ja `R` on se tyyppi, jonka konstruktorifunktio `c` palauttaa. Listarakenteella on siis kaksi tyyppiparametria.

Listan rakennuksessa päästään alkuun määrittelemällä `nil`-funktio:

```
nil = λX. λR. λc:X→R→R. λn:R. n;
```

Operaatio `cons` ottaa tyyppiä `X` olevan arvon (`hd`) ja listan (`tl`) ja palauttaa näiden yhdisteen. Häntälista instantioidaan "uudestaan" tyypillä `R`.

```
cons = λX. λhd:X. λtl:List X.
          λR. λc:X→R→R. λn:R. c hd (tl [R] c n);
```

Tämä näyttää kaikkien tyyppien kanssa melko vaikeaselkoiselta, mutta itse operaatio on yksinkertainen. Kokeilemalla näkee, että tuloksena on todellakin muotoa (2) oleva kaksiparametrinen funktio. Määrittelemällä tämä funktio `c` sopivasti saadaan

```
isnil = λX. λl:List X. l [Bool] (λhd:X. λtl:Bool. false) true;
```

Muutkin listaprimitiivit on mahdollista toteuttaa järjestelmä F:ssä, mutta tämä on hieman vaikeampaa. Pierce esittää kirjassaan myös, kuinka listaprimitiivien avulla voidaan toteuttaa esimerkiksi funktio, joka lajittelee parametrina annetun listan.

3.4 Ominaisuuksia

Järjestelmä F:llä on seuraavat keskeiset ominaisuudet:

- (Säilymisteoreema) Jos $\Gamma \vdash \mathtt{t} : \mathtt{T}$ ja $\mathtt{t} \to \mathtt{t}'$, niin $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. Tyypitetyn termin tyyppi siis säilyy evaluoinnissa.
- (Etenemisteoreema) Jos t on suljettu ja hyvin tyypitetty, niin joko t on arvo tai siihen voidaan soveltaa evaluointisääntöä $\mathtt{t} \to \mathtt{t}'$ jollakin $\mathtt{t}'$.
- (Normalisointi) Jokaisen hyvin tyypitetyn ohjelman evaluointi päättyy.

Nämä teoreemat ovat tuttuja jo yksinkertaisesta lambdakalkyylistä. Järjestelmä F:n laskentavoima on kuitenkin suuri: kuten edellä nähtiin, järjestelmässä voidaan toteuttaa jopa lajittelufunktio (vieläpä käyttämättä `fix`:iä). Normalisointiominaisuuden todistaminen onkin siten melkoinen saavutus.

Jotain kuitenkin menetetään: vuonna 1994 J. Wells osoitti, että tyyppinpäättely ei onnistu järjestelmä F:ssä. Tämä antaa aiheen kysyä, voisiko F:ää jotenkin rajoittaa, niin että rajoitetulle F:lle tyyppinpäättely toimisi.

Yksinkertaisin vaihtoehto on ML:n ja Haskellin käyttämä let-monimuotoisuus. Tästä käytetään myös termiä *prenex*-monimuotoisuus, mikä viittaa predikaattilogiikkaan. Tyyppimuuttujille sallitaan vain sellaiset tyyppiarvot, jotka eivät sisällä kvanttoreita (*monotyypit*), ja jossa kvantifioidut tyypit (*polytyypit* tai *tyyppiskeemat*) saavat esiintyä vain nuolioperaattoreiden oikealla puolella.

Let-monimuotoisuutta kutsutaan myös *predikatiiviseksi*, kun taas järjestelmä F:n monimuotoisuus on *impredikatiivista*. Näillä termeillä viitataan logiikkaan ja joukon käsitteen määrittelyyn. Jos järjestelmä F:ssä määritellään tyyppi $\mathtt{T} = \forall\mathtt{X}.\mathtt{X}\to\mathtt{X}$, tyyppimuuttujan X mahdollisia arvoja ovat *kaikki* tyypit, erityisesti myös T. (Tämä ei ole mahdollista ML:ssä.)

Toinen rajoitus järjestelmä F:lle on *toisen asteen* (rank-2) monimuotoisuus. Piirretään tyyppi T (syntaksi)puuna, jossa "operaattoreina" ovat kuvaus ($\to$) ja universaalikvanttori ($\forall$). Tyyppi T on astetta 2, jos millään polulla juuresta $\forall$-kvanttoriin ei mennä kuvausoperaattorin vasemmalle puolelle kahdesti tai useammin. Esimerkiksi tyyppi $\mathtt{Y} = (\forall\mathtt{X}.\mathtt{X}\to\mathtt{X})\to\mathtt{Nat}$ on astetta 2, mutta tyyppi $\mathtt{Y}\to\mathtt{Nat}$ ei ole.

4 Eksistentiaalityypit

Järjestelmä F lisää yksinkertaiseen lambdakalkyyliin universaalityypin $\forall\mathtt{X}.\mathtt{T}$. Universaalikvanttorin $\forall$ lisäksi voidaan kysyä, mitä käyttöä eksistentiaalikvanttorilla $\exists$ olisi.

4.1 Määritelmiä ja motivointia

Universaalityyppiä $\forall$X.T voi ajatella kahdella tapaa. *Looginen intuitio* sanoo, että tämän tyypin alkiot ovat arvoja, joiden tyyppi on [X $\mapsto$ S]T kaikilla mahdollisilla S. Tämä intuitio vastaa tyypinhävittämistä: monimuotoinen identiteettikuvaus (1) kuvautuu tyypittämättömäksi funktioksi λx.x, joka kuvaa minkä tahansa tyypin S alkion samalle tyypille. Toisaalta voidaan käyttää *operationaalista intuitiota*, jonka mukaan tyypin $\forall$X.T alkio on funktio, joka kuvaa tyypin S termiksi, jonka tyyppi on [X $\mapsto$ S]T. Tämä vastaa hyvin järjestelmä F:ää, jossa tyypinsovellukselle on oma evaluointisääntönsä. Universaalityypin sovellusta voi siis pitää tietynlaisena laskenta-askeleena.

Eksistentiaalityypille, jota merkitään symbolilla {$\exists$X, T}, voidaan myös käyttää kahta erilaista tulkintaa. Looginen intuitio sanoo, että tämän tyypin alkiot ovat tyyppiä [X $\mapsto$ S]T, *jollakin* tyypillä S. Operationaalisen intuition mukaan tyyppi S "tiedetään": kyseessä on taas kuvaus, eksistentiaalityypiltä jollekin tietylle tyypille S. Idea on se, että tyyppiä {$\exists$X, T} voidaan käyttää tietämättä, mikä S todellisuudessa on. Tietämys on upotettu tyyppiin itseensä.

Eksistentiaalityyppiä voi ajatella *moduulina*, joka peittää tyypin S näkyvistä ja tarjoaa abstraktin rajapinnan sen käyttöön. Tämä intuitio konkretisoidaan määritelmällä: tyypin {$\exists$X,T} alkio on pari {*S, t}, missä S on (esitys)tyyppi ja t termi, jonka tyyppi on [X $\mapsto$ S]T. Siis

- tyyppi T on abstrakti tyyppi, jollaisena moduuli näkyy "ulospäin" käyttäjilleen;
- tyyppi S on moduulin esitystyyppi, jota moduuli käyttää toteutuksessaan. Se ei näy moduulin käyttäjille;
- termi t on moduulin *ohjelmointirajapinta*. Se määrittelee, mitä operaatioita tyypille T voidaan tehdä, ja toteuttaa nämä käyttäen tyyppiä S.

Tarkastellaan seuraavaksi esimerkkejä. Moduulilla

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}}
```

on eksistentiaalityyppi {$\exists$X, {a:X, f:X→X}}. Tyyppi ei kuitenkaan ole yksikäsitteinen, sillä f:n tyyppi voisi olla myös X→Nat jollakin X (tässä tapauksessa X on Nat). Tämä tarkoittaa, että moduulin kirjoittajan täytyy askriptiolla kertoa, mikä moduulin tyypin tulee olla:

```
  p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}}
▸ p : {∃X, {a:X, f:X→Nat}}
```

Tämä ilmaistaan muodollisesti tyypityssäännöllä

$$\frac{\Gamma \vdash \mathtt{t}_2 : [\mathtt{X} \mapsto \mathtt{U}]\mathtt{T}_2}{\Gamma \vdash \{\mathtt{*U},\mathtt{t}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}_2\} : \{\exists\mathtt{X},\mathtt{T}_2\}} \qquad \text{(T-PACK)}$$

Esitystyyppi ei vaikuta eksistentiaalityyppiin (eikä tietysti pidäkään), joten moduulilla

```
  q = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}}
▸ q : {∃X, {a:X, f:X→Nat}}
```

on sama tyyppi kuin p:llä.

4.2 Eksistentiaalityypin käyttö

Jotta eksistentiaalityypeistä olisi mitään hyötyä, täytyy vielä määritellä, kuinka niitä voidaan käyttää. Tämä täytyy tehdä jollain tapaa kontrolloidusti, jotta moduulin esitystyyppi pysyy näkymättömissä.

Tähän tarkoitukseen käytetään `let`-rakennetta, jolla annetaan moduulin tyyppi- ja termikomponenteille paikallinen nimi:

```
  let {X,x}=p in (x.f x.a);
▸ 6 : Nat
  let {X,x}=q in (x.f x.a);
▸ 0 : Nat
```

Rakenteen tyypityssääntö on

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\exists\mathtt{X}, \mathtt{T}_{12}\} \qquad \Gamma, \mathtt{X}, \mathtt{x}{:}\mathtt{T}_{12} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \qquad \text{(T-UNPACK)}$$

Jos siis $\mathtt{t}_1$ on lauseke, joka on eksistentiaalityyppiä, niin sen tyyppi- ja termikomponentit voidaan sitoa nimiin `X` ja `x`, ja näitä käyttäen voidaan evaluoida termi $\mathtt{t}_2$.

Koska esitystyyppi on abstrakti, on selvästikin mahdollista soveltaa `x`:ää ainoastaan niillä tavoin, jotka moduulin rajapinta (eksistentiaalityyppi) mahdollistaa. Yllä mainitussa esimerkissä termi `succ(x.a)` ei tyypity, vaikka `p`:n esitystyyppi onkin `Nat`. Toinen tärkeä huomio on, että vaikka tyyppiä `X` käytetään termin $\mathtt{t}_2$ tyypin selvittämiseksi, se ei esiinny lopputuloksessa. Tyyppi $\mathtt{T}_2$ ei voi siis sisältää `X`:ää vapaana. Lause

```
  let {X,x}=p in x.a;
```

ei siksi ole tyypittyvä, koska tulostyyppi `X` ei näy `let`-rakenteen ulkopuolelle.

Eksistentiaalityyppien evaluointisääntö on yhteenveto jo esitetystä:

$$\begin{array}{l}\mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\mathtt{=}(\{\mathtt{*}\mathtt{T}_2,\mathtt{v}_{12}\}\ \mathtt{as}\ \mathtt{T}_1)\ \mathtt{in}\ \mathtt{t}_2 \\ \qquad \to [\mathtt{X} \mapsto \mathtt{T}_2][\mathtt{x} \mapsto \mathtt{v}_{12}]\mathtt{t}_2\end{array} \qquad \text{(E-UNPACKPACK)}$$

Jos siis `let`-lauseen ensimmäinen osa on redusoitu moduuliksi, niin sen sisällölle annetaan nimet `X` ja `x`, jonka jälkeen termi $\mathtt{t}_2$ evaluoidaan.

4.3 Tiedon abstrahointi

Eksistentiaalityyppejä voidaan käyttää ohjelmoitaessa moduuleita. Tarkastellaan *laskurityyppiä* esimerkkinä siitä kuinka tämä toimii käytännössä. Jos käytössä on ohjelmointikieli, joka tukee perinteistä tiedon abstrahointi -periaatetta, tavoitteena on määritellä *abstrakti tietotyyppi* (ATT), joka koostuu seuraavista elementeistä:

- Tyypillä täytyy olla nimi; olkoon se `Counter`.
- Laskuri täytyy toteuttaa muita (yksinkertaisempia) tyyppejä käyttäen. Nämä voivat olla muita abstrakteja tietotyyppejä tai (lopulta) kielen primitiivityyppejä. Tässä tapauksessa esitystyyppi voi olla `Nat`.
- Tarvitaan joukko operaatioita (abstraktiorajapinta), jotka mahdollistavat laskurin käsittelyn. Nämä (ja vain nämä) operaatiot tietävät, mikä on ATT:n esitystyyppi.

Kuten huomataan, tässä on suora vastaavuus eksistentiaalityyppeihin. Laskurimoduuli voidaan nyt kirjoittaa helposti:

```
counterPkg = {*Nat,
              {new = 1,
               get = λi:Nat i,
               inc = λi:Nat. succ(i)}} as
{∃Counter, {new: Counter,
            get: Counter→Nat,
            inc: Counter→Counter}};
```

(Tämän luvun esimerkit käyttävät puhtaasti funktionaalista ohjelmointityyliä, eikä laskurilla sen vuoksi ole tilamuuttujaa.) Tässä on määritelty sekä moduulin toteutus että sen abstraktiorajapinta. Jälkimmäinen määrää esimerkiksi sen, että `inc` ei palauta lukua, vaan tyypin `Counter`. Vasta `get`-operaation tulos on tyyppiä `Nat`.

Moduuli *avataan* ohjelman käyttöön `let`-rakenteella:

```
   let {Counter,c} = counterPkg in c.get (c.inc c.new);
▸ 2 : Nat
```

Tämä johtaa tietysti siihen, että muusta ohjelmasta tulee `let`-lauseen runko. Rungossa voidaan tyyppiä `Counter` kuitenkin käyttää aivan kuin se olisi primitiivityyppi: voidaan määritellä sitä käyttäviä funktioita ja jopa uusia tietotyyppejä, jotka taas otetaan käyttöön sisäkkäisellä `let`-rakenteella.

Abstraktin tietotyypin asemesta voitaisiin laskuri toteuttaa myös *oliona*. Inkrementointi toteutetaan lähettämällä oliolle `inc`-viesti, minkä seurauksena olio palauttaa uuden laskuriolion, jossa laskuria on päivitetty.

Määritellään eksistentiaalinen tyyppi

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};
```

ja laskuriolio

```
c = {*Nat, {state = 5,
            methods = {get = λx:Nat. x,
                       inc = λx:Nat. succ(x)}}}
    as Counter;
```

Jokainen olio on siis eksistentiaalinen moduuli, joka sisältää sekä olion tilan että metodit. Edellä eksistentiaalisuutta käytettiin antamaan uusi nimi `Counter` moduulin esitystyypille. Tässä tapauksessa taas jokainen olio sisältää omat metodinsa.

Olion tilaan päästään käsiksi helposti avaamalla oliomoduuli ja kysymällä:

```
  let {X,body} = c in body.methods.get(body.state);
▸ 5: Nat
```

Inkrementointi ei kuitenkaan onnistu näin helposti. Metodin `inc` paluutyyppi on moduulin esitystyyppi `X`, joka johtaa tyypitysvirheeseen, kuten edellä nähtiin. Ongelma ratkaistaan paketoimalla tyyppiä `X` oleva arvo uudeksi laskuriobjektiksi

```
c′ = let {X,body} = c in
      {*X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
     as Counter
```

Tämä operaatio voidaan edelleen helposti kirjoittaa funktioksi, joka ottaa parametrikseen laskurin ja palauttaa inkrementoidun laskuriolion.

5 Yhteenveto

Lisättäessä yksinkertaisesti tyypitettyyn lambdakalkyyliin parametrinen monimuotoisuus saadaan järjestelmä F, joka on elegantti, laskentavoimainen ja paljon tutkittu kalkyyli. Siinä otetaan käyttöön kvantifioidut, monimuotoiset tyypit. Sovellettaessa tällaista monimuotoista tyyppiä se instantioidaan jollain toisella tyypillä.

Eksistentiaalikvanttoria voidaan käyttää datan abstrahointiin, jossa moduulin esitystyyppi peitetään moduulin käyttäjän näkyvistä. Osoittautuu myös [Pie02, §24.3], että kaikki eksistentiaalityyppeihin liittyvä on koodattavissa universaalityyppien ja tyypinsovelluksen avulla. Tässä mielessä eksistentiaalityypit eivät tuo järjestelmä F:ään mitään uutta.

Järjestelmä F toimii pohjana, jonka päälle voidaan jatkossa lisätä uusia mekanismeja, jolloin saadaan kehittyneempiä kalkyylejä. Tällaisia laajennuksia ovat esimerkiksi alityypit ($F_{<:}$), tyyppioperaattorit (F_ω) ja näiden yhdistelmä, järjestelmä $F^\omega_{<:}$.

Lähteet

CaW85 Cardelli, L., Wegner., P., On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17,4(1985), sivut 471–522.

Pie02 Pierce, B., *Types and programming languages.* MIT Press, 2002.