Tyyppitoimitukset ja alityypitys

Vesa Karvonen

Helsinki 7.5.2003

Tyyppiteoria ja ohjelmointikielet –seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Tämä kirjoitelma käsittelee korkeamman kertaluvun alityypitystä sekä puhtaan funktionaalisia olioita, noudattaen orjallisesti Benjamin C. Piercen kirjan *Types and Programming Languages* lukuja 31 ja 32 [Pie02].

Kirjoitelman ensimmäisessä osassa tutustumme lyhyesti tyyppisysteemiin $F^{\omega}_{<:}$ (lausutaan "F–omega–ali"), joka muodostaa pohjan tekstin jälkimmäisessä osassa suhteellisen laajoilla esimerkeillä käsiteltävälle puhtaan funktionaaliselle oliomallille.

Kirjoitelmassa käytetyt notaatiot vastaavat pitkälti kirjan [Pie02], sekä seminaarissa aiemmin ilmestyneiden kirjoitelmien, notaatioita. Yksinkertaistaen voitaisiin sanoa, että tyypit kirjoitetaan isoilla ja termit pienillä kirjaimilla.

2 Korkeamman kertaluvun alityypitys

Tutustutaan aluksi systeemiin $F^{\omega}_{<:}$, joka voidaan nähdä systeemin $F_{<:}$ laajennoksena, eli toisen kertaluokan polymorfisena lambda–kalkyylinä rajoitetulla kvantifioinnilla sekä tyyppioperaattoreilla. Erityisesti alityypitysrelaatiota laajennetaan lajia (kind) $*$ olevista tyypeistä korkeamman lajisiin tyyppeihin.

2.1 Alityypityksen ja tyyppioperaattorien yhdistäminen

Alityypityksen ja rajoitetun kvantifioinnin yhdistäminen tyyppioperaattorien kanssa pakottaa tekemään useita valintoja yhdistetyn systeemin määrittelyssä.

Alityypityksen kanssa muotoa $\lambda X :: K_1.T_2$ olevat tyyppioperaattorit voitaisiin laajentaa muotoa $\lambda X <: T_1.T_2$ oleviin *rajoitettuihin tyyppioperaattoreihin*. Yksinkertaisuuden vuoksi kuitenkin tässä määrittelemämme systeemi sisältää rajoitetun kvantifioinnin, mutta *rajoittamattomat* tyyppioperaattorit.

Alityypitysrelaation laajentaminen tyyppioperaattoreilla voidaan myös tehdä useilla tavoilla. Valitsemme jälleen yksinkertaisimman vaihtoehdon, jossa alityypitysrelaatio laajennetaan vastinkohdittain (pointwise) tyyppioperaattoreihin. Siis abstraktio $\lambda X.S$ on abstraktion $\lambda X.T$ alityyppi, mikäli kaikilla argumenteilla U, sovelluksen $(\lambda X.S)\ U$ tyyppi on sovelluksen $(\lambda X.T)\ U$ tyypin alityyppi. Tämä voidaan esittää seuraavana sääntönä:

$$\frac{\Gamma, X \vdash S <: T}{\Gamma \vdash \lambda X.S <: \lambda X.T} \qquad \text{(S-ABS)}$$

Vastaavasti, jos F ja G ovat tyyppioperaattoreita, joille pätee $F <: G$, niin silloin $F\ U <: G\ U$. Tämän esittävä sääntö

$$\frac{\Gamma \vdash F <: G}{\Gamma \vdash F\ U <: G\ U} \qquad \text{(S-APP)}$$

pätee vain kuin F ja G saavat saman argumentin U, koska tieto siitä, että F on G vastinkohdittainen alityyppi, ei kerro mitään kyseisten tyyppioperaattoreiden käyttäytymisestä eri argumenteilla.

Jos tyypit S ja T ovat ekvivalentteja, niin silloin ne ovat luonnollisesti toistensa alityyppejä. Tästä saamme vielä seuraavan alityypityssäännön:

$$\frac{\Gamma \vdash \mathrm{S} :: \mathrm{K} \qquad \Gamma \vdash \mathrm{T} :: \mathrm{K} \qquad \mathrm{S} \equiv \mathrm{T}}{\Gamma \vdash \mathrm{S} <: \mathrm{T}} \tag{S-EQ}$$

Alityypityksen laajentaminen edelliseen tapaan lajista $*$ lajiin $*\Rightarrow*$ toimii myös monimutkaisemmille lajeille. Esimerkiksi, jos P ja Q ovat lajia $*\Rightarrow*\Rightarrow*$, niin sanomme, että P <: Q, mikäli kaikille U, sovellus P U on sovelluksen Q U alityyppiä lajissa $*\Rightarrow*$.

Hyödyllisenä sivuvaikutuksena edellisille määritykselle on, että kaikkien korkeampien lajien alityyppirelaatioilla on maksimaalinen elementti. Jos määritämme, että $\mathrm{Top}[*] = \mathrm{Top}$ ja

$$\mathrm{Top}[\mathrm{K}_1 \Rightarrow \mathrm{K}_2] \stackrel{\mathrm{def}}{=} \lambda \mathrm{X} :: \mathrm{K}_1.\mathrm{Top}[\mathrm{K}_2],$$

niin silloin $\Gamma \vdash \mathrm{S} <: \mathrm{Top}[\mathrm{K}]$, kun S on lajia K.

$\mathrm{F}^{\omega}_{<:}$ perii $\mathrm{F}_{<:}$:sta muotoa $\forall \mathrm{X} <: \mathrm{T}_1.\mathrm{T}_2$ olevat rajoitetut kvanttorit, sekä F_{ω}:sta muotoa $\forall \mathrm{X} :: \mathrm{K}_1.\mathrm{T}_2$ olevat korkeamman kertaluvun rajoittamattomat kvanttorit. Nämä voidaan sisällyttää sellaisinaan $\mathrm{F}^{\omega}_{<:}$:iin, kunhan pidetään vain mielessä, että T_1 voi siis olla varsinaisen tyypin lisäksi myös tyyppioperaattori, sekä ajatellaan $\forall \mathrm{X} :: \mathrm{K}_1.\mathrm{T}_2$ lyhenteeksi muotoa $\forall \mathrm{X} <: \mathrm{Top}[\mathrm{K}_1].\mathrm{T}_2$ olevalle tyypille.

$\mathrm{F}_{<:}$:n tavoin $\mathrm{F}^{\omega}_{<:}$:sta on kaksi erilaista variaatiota: yksinkertaisempi *ydin*–$\mathrm{F}^{\omega}_{<:}$ sekä ilmaisuvoimaisempi *täysi*–$\mathrm{F}^{\omega}_{<:}$. Jatkossa keskitymme ydin variaatioon. Täysi variaatio olisi myös semanttisesti mielekäs, mutta sen metateoreettisia ominaisuuksia ei ole toistaiseksi selvitetty [Pie02].

2.2 Määrityksestä

$\mathrm{F}^{\omega}_{<:}$:n koko määritys löytyy kirjan [Pie02]:n sivuilta 470 ja 471.[1] Huomattava piirre määrityksessä on, että siinä on kahdenlaisia tyyppimuuttujien sidontoja: X :: K tyyppioperaattoreissa ja X <: T kvanttoreissa. Ainoastaan jälkimmäinen muoto sallitaan konteksteissa. Kun X :: K muotoinen sidonta siirtyy kontekstiin, kuten esimerkiksi säännössä

$$\frac{\Gamma, \mathrm{X} <: \mathrm{Top}[\mathrm{K}_1] \vdash \mathrm{T}_2 :: \mathrm{K}_2}{\Gamma \vdash \lambda \mathrm{X} :: \mathrm{K}_1.\mathrm{T}_2 :: \mathrm{K}_1 \Rightarrow \mathrm{K}_2} \tag{K-ABS}$$

muutetaan se muotoon X <: Top[K]. Toinen huomattava piirre on, että $\mathrm{F}_{<:}$:n sääntö S-REFL ja F_{ω}:n sääntö T-EQ jätetään kokonaan pois, koska niiden piirteet seuraavat säännöistä S-EQ ja Q-REFL sekä T-SUB ja S-EQ, vastaavasti.

3 Puhtaan funktionaaliset oliot

Jatkamme seuraavaksi eksistentiaalisen oliomallin kehitystä, johon tutustuimme lyhyesti tyyppikvanttoreiden sekä tyyppikvanttoreiden ja alityypityksen yhteydessä. Tarkastelimme

[1] Määrityksen pituuden takia sen jäljentäminen tähän ei olisi mielekästä. Tuon sivuista kopioita esityksen seuraamisen helpottamiseksi.

silloin miten eksistentiaalityypeillä voitiin esittää abstrakteja tietotyyppejä sekä olioita. Katsotaan vielä kertauksen vuoksi jo aiemmin nähtyjä `Counter`–olion määrityksiä.

`Counter`–olion tyyppi oli muotoa

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

Vastaavasti `Counter`–tyypin mukaisen olion toteutus oli muotoa

```
c = {*CounterR,
     {state = {x=5},
      methods = {get = λr:CounterR. r.x,
                 inc = λr:CounterR. {x=succ(r.x)}}}} as Counter;
▶ c : Counter
```

kun käytämme lyhennettä `CounterR` olion sisäiselle esitysmuodolle:

```
CounterR = {x:Nat};
```

`Counter`–tyyppisen olion metodien kutsumiseksi olio on ensin avattava ja jos haluamme palauttaa lopuksi olion, on eksplisiittisesti luotava alkuperäisen muotoinen olio.

```
sendinc = λc:Counter,
            let {X,body} = c in
              {*X,
               {state = body.methods.inc(body.state),
                methods = body.methods}} as Counter;
▶ sendinc : Counter → Counter
```

Edellisessä siis `let {X,body} = c in ...` avaa olion ja vastaavasti `{*X,{...}} as Counter` luo uuden olion.

3.1 Alityypitys

Seurauksena eksistentiaalityyppien alityypityssäännöstä

$$\frac{\Gamma, X <: U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X <: U, S_2\} <: \{\exists X <: U, T_2\}} \qquad \text{(S-SOME)}$$

olioidemme koodaus saa suoraan odottamamme alityypityksen olioiden tyyppien välille. Voimme siis esim. koodata olion, jolla on enemmän metodeja kuin `Counter`–oliolla, ja sen tyyppi on suoraan `Counter`–tyypin alityyppi:

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

rc = {*CounterR,
      {state = {x=0},
       methods = {get = λr:CounterR. r.x,
                  inc = λr:CounterR. {x=succ(r.x)},
                  reset = λr:CounterR. {x=0}}}} as ResetCounter;
▶ rc : ResetCounter
```

Nyt siis voimme antaa `ResetCounter`–tyyppiä olevan olion aiemmin `Counter`–tyyppiä oleville olioille määrittelemillemme funktioille, kuten `sendinc`:

```
  rc1 = sendinc rc;
▶ rc1 : Counter
```

Ikävä kyllä tällöin menetämme tyyppi–informaatiota, sillä nyt `rc1`:n tyyppi on `Counter` eikä `ResetCounter`.

3.2 Rajoitettu kvantifiointi

Voisi olettaa, että rajoitetun kvantifioinnin avulla voisimme kirjoittaa `sendinc`–funktiosta tyyppiä $\forall$`C<:Counter. C→C` olevan muodon, jolloin siis emme menettäisi tyyppi–informaatiota. On kuitenkin todistettavissa, että muotoa $\forall$`C<:Counter. C→C` olevat tyypit sisältävät systeemissä $F_{<:}$ *vain* identiteettifunktioita. Määrityksestä

```
  sendinc =
     λC<:Counter. λc:C.
       let {X,body} = c in
         {*X,
          {state = body.methods.inc(body.state),
           methods = body.methods}}
         as C;
▶ Error: existential type expected
```

saamamme virheilmoitus johtuu siitä, että viimeisellä rivillä `C` on tyyppimuuttuja eikä eksistentiaalityyppi. Tämä rajoitus on myös olennainen, eikä pelkästään tyyppitarkastajan mielivaltainen rajoitus, sillä ilman sitä edelliselle funktiolle voisi antaa tyyppiä

```
 JunkCounter =
   {∃X, {state:X, methods:{get:X→Nat,inc:X→X}, junk:Bool}}
```

olevan olion, jolloin tuloksena olisi virheellisesti `JunkCounter`–tyyppiä oleva olio, josta puuttuisi siis `junk`–kenttä.

Seuraavaksi tarkastelemme miten tähän ongelmaan voidaan puuttua $F^{\omega}_{<:}$–systeemissä.

3.3 Rajapintatyypit

Tyyppioperaattoreiden avulla voimme pilkkoa `Counter`–tyypin esityksen kahdeksi tyyppioperaattoriksi: geneeriseksi `Object`–tyypiksi

```
  Object = λM::*⇒*. {∃X, {state:X, methods:M X}};
```

sekä spesifiseksi `CounterM`–tyypiksi

```
  CounterM = λR. {get: R→Nat, inc:R→R};
```

joilla saamme muodostettua `Counter`–tyypin soveltamalla lajia `(*⇒*)⇒*` olevaa `Object`–tyyppiä, lajia `*⇒*` olevaan `CounterM`–tyyppiin:

```
  Counter = Object CounterM;
```

Ideana on siis erottaa oliotyypistä toiseen vaihtelevat metodien rajapinnat, sekä olion pakkaaminen eksistentiaalityypiksi toisistaan. Voimme paloitella `ResetCounter`–tyypin vastaavasti

```
  ResetCounterM = λR. {get: R→Nat, inc:R→R, reset:R→R};
  ResetCounter = Object ResetCounterM;
```

Tällöin pätee

```
ResetCounter <: Counter
ResetCounterM <: CounterM
```

Saavutamme siis olion tyypin jakamisella geneeriseen `Object`–tyyppiin, sekä spesifisiin rajapintatyyppeihin, mielekkään rajapintojen alityypityksen.

3.4 Viestien lähettäminen olioille

Voimme nyt ilmasta `sendinc`–funktion siten, että se ei kadota tyyppi–informaatiota.

```
sendinc =
  λM<:CounterM. λc:Object M.
    let {X, b} = c in
      {*X,
       {state = b.methods.inc(b.state),
        methods = b.methods}}
       as Object M;
▶ sendinc : ∀M<:CounterM. Object M → Object M
```

Jos vertaa tätä aiempaan virheelliseen `sendinc`–funktion määritelmään, niin on huomattava, että `Object`–tyyppi olennaisesti rajoittaa olioiden rakennetta, eikä olioihin siis voi enää lisätä roskaa (junk).

Kutsuessamme edellisen kaltaisia polymorfisia funktioita, annamme niille sekä rajapintatyypin, että olion, joka toteuttaa kyseisen rajapinnan, jolloin funktio osaa rakentaa olion, jolla on oikea tyyppi.

```
sendget [ResetCounterM]
            (sendreset [ResetCounterM]
               (sendinc [ResetCounterM] rc));
▶ 0 : Nat
```

Huomaa, että yllä sekä `sendreset` että `sendinc` kutsut säilyttävät olion tyypin.

3.5 Yksinkertaiset luokat

Kun oletamme, että kaikki oliot käyttävät samaa sisäistä esitysmuotoa, yksinkertainen luokka voidaan nähdä pelkkänä tietueena metodeista:

```
counterClass =
   {get = λr:CounterR. r.x,
    inc = λr:CounterR. {x=succ(r.x)}}
 as CounterM CounterR;
▶ counterClass : CounterM CounterR
```

Luokan tehtävänä on siis määritellä olion metodit. Yhdistämällä tilan ja luokan samaan pakkaukseen saamme olion:

```
c = {*CounterR,
     {state = {x=0},
      methods = counterClass}}
     as Counter;
```

```
▶ c : Counter
```

Uuden luokan määrittäminen onnistuu yksinkertaisesti määrittelemällä uusi tietue metodeista:

```
  resetCounterClass =
     let super = counterClass in
     {get = super.get,
      inc = super.inc,
      reset = λr:CounterR. {x=0}}
     as ResetCounterM CounterR;
▶ resetCounterClass : ResetCounterM CounterR
```

Emme kuitenkaan vielä pysty lisäämään olioon aliluokissa uusia instanssimuuttujia tai viittaamaan luokan toisiin metodeihin rekursiivisesti muuttujalla.

3.6 Polymorfinen päivitys

Toistaiseksi laskureidemme ainoa sisäinen esitysmuoto on ollut `CounterR`. Haluaisimme mahdollisuuden laajentaa tätä esitysmuotoa aliluokissa, joka tarkoittaa sitä, että yliluokkien on oltava polymorfisia kyseisen esitysmuodon suhteen, eli luokkien metodien on oltava polymorfisia saamansa esitysmuodon suhteen. Nythän esim. `inc`–metodi

```
  inc = λr:CounterR. {x=succ(r.x)}
▶ inc : CounterR → CounterR
```

on monomorfinen. Jos yritämme laajentaa `inc`–metodin suoraviivaisesti tyyppiä

```
  inc : ∀S<:{x:Nat}.S→S
```

olevaksi polymorfiseksi metodiksi, ajaudumme vastaavaan ongelmaan, kuin aiemmin virheellisen `sendinc`–metodin kanssa, kun se kadotti `junk`–kentän.

Yksinkertainen, muttei ainoa, tapa ratkaista tämä ongelma on määritellä polymorfinen tietueen päivitysoperaatio, jolla siis voidaan päivittää tietueen kenttää, tietämättä minkä tyyppinen tietue täsmälleen on. Tällöin voisimme ohjelmoida polymorfisen `inc`–metodin suurin piirtein seuraavasti:

```
  f = λX<:{a:Nat}. λr:X. r←a = succ(r.a);
```

Tässä on kuitenkin oltava varovaisia, koska naivi sääntö päivitysoperaation tyypitykselle

$$\frac{\Gamma \vdash \mathrm{r} : \mathrm{R} \qquad \Gamma \vdash \mathrm{R} <: \{\mathrm{l}_j : \mathrm{T}_j\} \qquad \Gamma \vdash \mathrm{t} : \mathrm{T}_j}{\Gamma \vdash \mathrm{r} \leftarrow \mathrm{l}_j = \mathrm{t} : \mathrm{R}}$$

on kelvoton! Esimerkiksi lauseke

```
  {x={a=5,b=6},y=true}←x={a=8}
```

antaisi väärän tuloksen

```
▶ {x={a=8},y=true} : {x:{a:Nat,b:Nat},y:Bool}
```

koska `{x:{a:Nat,b:Nat},y:Bool} <: {x:{a:Nat}}`.

Ongelman aiheuttaa syvyyssuuntainen alityypitys `x`–kentän suhteen. Ongelman voi siis välttää kieltämällä syvyyssuuntaisen alityypityksen kentiltä, joita halutaan päivittää. Lausekkeissa päivitettävät kentät sitten ilmaistaisiin `#`–merkein. Esimerkiksi edellinen `f`–funktio kirjoitettaisiin

```
  f = λX<:{#a:Nat}. λr:X. r←a = succ(r.a);
▶ f : ∀X<:{#a:Nat}. X → X
```

Edelliselle funktiolle on siis annettava tietua, jossa on `a`–kentän on oltava päivitettävä.

```
  r = {#a=0, b=true};
  f [{#a:Nat,b:Bool}] r;
▶ {#a=1, b=true} : {#a:Nat, b:Bool}
```

Katsotaan vielä polymorfiseen päivitykseen liittyviä keskeisiä[2] määrityksiä. Ensinnäkin tietuiden syntaksia laajennetaan merkinnällä, joka kertoo onko kyseessä invariantti päivitettävä kenttä vai kovariantti kiinteä kenttä:

$$
\begin{array}{lcll}
\mathrm{t} & ::= & \ldots & \text{termit:} \\
 & & \{\iota_i \mathrm{l}_i = \mathrm{t}_i{}^{i\in 1..n}\} & \text{tietue} \\
 & & \mathrm{t} \leftarrow \mathrm{l} = \mathrm{t} & \text{kentän päivitys} \\
 & & & \\
\mathrm{T} & ::= & \ldots & \text{tyypit:} \\
 & & \{\iota_i \mathrm{l}_i : \mathrm{T}_i{}^{i\in 1..n}\} & \text{tietueen tyyppi} \\
 & & & \\
\iota & ::= & \# & \text{invariantti kenttä} \\
 & & \epsilon & \text{kovariantti kenttä}
\end{array}
$$

Itse funktionaalisen päivitysoperaation määrittää sääntö

$$
\begin{array}{c}
\{\iota_j \mathrm{l}_j = \mathrm{v}_j{}^{j\in 1..n}\} \leftarrow \mathrm{l}_i = \mathrm{v} \\
\longrightarrow \{\iota_j \mathrm{l}_j = \mathrm{v}_j{}^{j\in 1..i-1}, \iota_i \mathrm{l}_i = \mathrm{v}, \iota_k \mathrm{l}_k = \mathrm{v}_k{}^{k\in i+1..n}\}
\end{array}
\qquad \text{(E-UPDATE)}
$$

ja päivitysoperaation tyypityksen määrittää sääntö

$$
\frac{\Gamma \vdash \mathrm{r} : \mathrm{R} \qquad \Gamma \vdash \mathrm{R} <: \{\#\mathrm{l}_j : \mathrm{T}_j\} \qquad \Gamma \vdash \mathrm{t} : \mathrm{T}_j}{\Gamma \vdash \mathrm{r} \leftarrow \mathrm{l}_j = \mathrm{t} : \mathrm{R}} \qquad \text{(T-UPDATE)}
$$

Kuten jo aiemmin vihjattiin, tietuiden alityypityssääntöä muutetaan siten, että vain kovarianttien, merkitsemättömien, kenttien alityypitys sallitaan:

$$
\frac{\begin{array}{c}\text{for each } i \quad \Gamma \vdash \mathrm{S}_i <: \mathrm{T}_i \\ \text{if } \iota_i = \#, \text{ then } \Gamma \vdash \mathrm{T}_i <: \mathrm{S}_i\end{array}}{\Gamma \vdash \{\iota_i \mathrm{l}_i : \mathrm{S}_i{}^{i\in 1..n}\} <: \{\iota_i \mathrm{l}_i : \mathrm{T}_i{}^{i\in 1..n}\}} \qquad \text{(S-RCDDEPTH)}
$$

Lisäksi lisätään vielä sääntö, joka sallii päivitysmerkinnän pudottamisen pois tietueesta:

$$
\Gamma \vdash \{...\#\mathrm{l}_i : \mathrm{S}_i ...\} <: \{...\mathrm{l}_i : \mathrm{S}_i ...\} \qquad \text{(S-RCDVARIANCE)}
$$

Edellinen sääntö on turvallinen, koska päivitysoperaation tyypityssääntö vaatii kyseisen merkinnän, joten merkinnän pudottaminen pois ei mahdollista virheellistä tyypitystä.

[2] Laajennos vaikuttaa muihin kuin tässä esitettyihin määrityksiin lähinnä siten, että tietuiden mukana kuljetetaan merkintää kenttien päivitettävyydestä.

3.7 Instanssimuuttujien lisääminen

Polymorfista päivitystä käyttäen voimme nyt toteuttaa `counterClass`–luokan, joka on polymorfinen sisäisen esitysmuotonsa suhteen.

```
CounterR = {#x:Nat};
counterClass =
   λR<:CounterR.
      {get = λs:R. s.x,
       inc = λs:R. s←x=succ(s.x)}
      as CounterM R;
▶ counterClass : ∀R<:CounterR. CounterM R
```

Luodaksemme olion `counterClass`–luokasta, annamme sille parametrina sisäisen esitysmuodon tyypin `CounterR`.

```
c = {*CounterR,
     {state = {#x=0},
      methods = counterClass [CounterR]}}
    as Object CounterM;
▶ c : Counter
```

Huomaa, että luodulla oliolla on tyyppi `Counter`, kuten aikaisemminkin. Muutokset instanssimuuttujien käsittelyyn ovat täysin luokkien sisäinen asia.

Voimme nyt kirjoittaa haluamamme kaltaiset luokat `resetCounterClass` ja `backupCounterClass`.

```
resetCounterClass =
   λR<:CounterR.
     let super = counterClass [R] in
       {get = super.get,
        inc = super.inc,
        reset = λs:R. s←x=0}
       as ResetCounterM R;
▶ resetCounterClass : ∀R<:CounterR. ResetCounterM R
```

```
BackupCounterM = λR.{get:R→Nat,inc:R→R,reset:R→R,backup:R→R};
BackupCounterR = {#x:Nat,#old:Nat};
backupCounterClass =
   λR<:BackupCounterR.
   let super = resetCounterClass [R] in
    {get = super.get,
     inc = super.inc,
     reset = λs:R. s←x=x.old,
     backup = λs:R. s←old=s.x}
    as BackupCounterM R;
▶ backupCounterClass : ∀R<:BackupCounterR. BackupCounterM R
```

Huomaa, että edelliset luokat käyttävät yliluokkiensa `get` ja `inc`–metodeja. Lisäksi `backupCounterClass` uudelleenmäärittelee `reset`–metodin, sekä lisää uuden instanssimuuttujan `old`.

3.8 Rekursiiviset luokat

Lopuksi tarkastelemmme vielä sellaisten luokkien määrittelyä, jotka voivat viitata omiin metodeihinsa. Kuten aiemminkin, rekursio saavutetaan `fix`–operaation avulla, jolloin luokka ottaa parametrina tietueen metodeja, jotka soveltuvat samalle sisäiselle esitysmuodolle:

```
counterClass =
   λR<:CounterR.
   λself: Unit→CounterM R.
   λ_:Unit.
      {get = λs:R. s.x,
       inc = λs:R. s←succ(s.x)}
      as CounterM R;
```

Huomaa, että luokan ydin on abstraktion sisällä, jotta sitä ei evaluoitaisi `fix`–operaatiossa, joka luo luokan metodit.

Oliota rakentaessa otamme siis kiintopisteen luokasta, ja annamme sille vielä `unit` parametrin, jotta saamme luokan metodit sisältävän tietueen.

```
c = {*CounterR,
     {state = {#x=0},
      methods = fix (counterClass [CounterR]) unit}}
    as Object CounterM;
▶ c : Counter
```

Toteutetaan vielä uusia ominaisuuksia käyttäviä luokkia. Ensiksi `setCounterClass`, jonka `inc`–metodi on toteutettu `SetCounterM`–rajapinnan määrittämän `set`–metodin avulla.

```
SetCounterM = λR. {get: R→Nat, set:R→Nat→R, inc:R→R};

setCounterClass =
   λR<:CounterR.
   λself: Unit→SetCounterM R.
   λ_:Unit.
      let super = counterClass [R] self unit in
      {get = super.get,
       set = λs:R. λn:Nat. s←x=n,
       inc = λs:R. (self unit).set s (succ((self unit).get s))}
   as SetCounterM R;
```

Huomaa edelleen miten `self`–funktiolle täytyy antaa `unit` parametri, jotta saadaan metodit sisältävä tietue.

Lopuksi vielä luokka, joka laskee montako kertaa sen `set`–metodia on kutsuttu:

```
InstrCounterM =
  λR. {get: R→Nat, set:R→Nat→R, inc:R→R, accesses:R→Nat};

InstrCounterR = {#x:Nat, #count:Nat};

instrCounterClass =
   λR<:InstrCounterR.
   λself: Unit→InstrCounterM R.
   λ_:Unit.
     let super = setCounterClass [R] self unit in
```

```
      {get = super.get,
       set = λs:R. λn:Nat.
               let r = super.set s n in
               r←count=succ(r.count),
       inc = super.inc,
       accesses = λs:R. s.count}
    as InstrCounterM R;
```

Luodaan vielä malliksi edellisen luokan olio, ja käytetään sitä:

```
  ic = {*InstrCounterR,
        {state = {#x=0,#count=0},
         methods = fix (instrCounterClass [InstrCounterR]) unit}}
       as Object InstrCounterM;
▶ ic : Object InstrCounterM

  sendaccesses [InstrCounterM] (sendinc [InstrCounterM] ic);
▶ 1 : Nat
```

Edellisessä `sendinc`–funktion toteutus on siis sama kuin aiemmin. `sendaccesses`–funktion toteutus on vastaavasti suoraviivainen.

4 Lopuksi

Lyhyesti sanottuna, olemme nähneet miten $F^{\omega}_{<:}$–systeemin avulla voidaan toteuttaa varsin monipuolinen puhtaan funktionaalinen oliomalli. Näkemämme oliomallin ominaisuuksiin kuuluu rajapintojen alityypitys sekä luokkien periminen. Pelkkään perintään (inheritance) perustuvista, oliokielistä poiketen, näkemämme oliomalli tukee $F^{\omega}_{<:}$–systeemin alityypitystä, parametrista polymorfismia sekä nimettömiä funktioita.

Lähteet

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