Rekursiiviset tyypit - teoria

Samppa Saarela

Helsinki 1999/04/14

Tyyppiteoria ja ohjelmointikielet - seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Tämä esitelmä käsittelee rekursiivisten tyyppien teoriaa perustuen Benjamin C. Piercen teokseen *Types and Programming Languages* (The MIT Press, 2002).

Listojen lisäksi on olemassa monia muitakin mielivaltaisen kokoisia tietorakenteita, joilla on kuitenkin yksinkertainen säännöllinen rakenne. Ei olisi mielekästä toteuttaa näitä kaikkia kielen primitiiveinä, vaan tarvitaan yleinen mekanismi näiden rakenteiden määrittelemiseksi tarpeen vaatiessa.

Esimerkiksi lista, joka sisältää numeroita määritellään joko tyhjäksi (nil) tai pariksi, joka koostuu luonnollisesta luvusta (Nat) ja toisesta listasta:

```
NatList = <nil:Unit, cons:{Nat,NatList}>;
```

Tämän yhtälö muuntuu määritelmäksi siirtämällä rekursio =-merkin oikealle puolelle. Tällainen määritelmä saadaan aikaiseksi uudella tyyppien määrittelemiseen tarkoitetulla rekursio-operaattorilla μ:

```
NatList = μX. <nil:Unit, cons:{Nat,X}>;
```

Määritelmä tarkoittaa ”NatList on ääretön tyyppi, joka toteuttaa yhtälön `X = <nil:Unit, cons:{Nat,X}>`”.

2 Esimerkkkejä

Tässä luvussa käsitellään rekursiivisia tyyppejä esimerkkien valossa.

2.1 Listojen operaatiot

Listojen käsittelemiseen tarvitaan vakio *nil*, konstruktori *cons*, jolla lisätään listan alkuun uusi elementti, *isnil* operaattori, joka kertoo, onko lista tyhjä sekä destruktorit *hd* ja *tl*, jotka palauttavat joko listan ensimmäisen alkion (*hd*) tai loppulistan (*tl*). Nämä funktiot määritellään seuraavasti:

```
    nil = <nil=unit> as NatList;
▶   nil : NatList

    cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList;
▶   cons : Nat → NatList → NatList

    isnil = λl:NatList. case l of
                        <nil=u> ⇒ true
                      | <cons=p> ⇒ false
▶   isnil : NatList → Bool
```

```
    hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1;
▶   hd : NatList → Nat

    tl = λl:NatList. case l of <nil=u> ⇒ | <cons=p> ⇒ p.2;
▶   tl : NatList → NatList
```

Näiden määritelmien avulla voidaan esimerkiksi määritellä rekursiivinen funktio, `sumlist`, joka laskee kaikkien listan elementtien summan. Rekursiivisten tyyppien ilmentymät määritellään käyttämällä `fix`-funktiota.

```
    sumlist = fix (λs:NatList→Nat. λl:NatList.
                    if isnil l then 0 else plus (hd l) (s (tl l)));
▶   sumlist : NatList → Nat
```

2.2 Nälkäiset funktiot ja virrat

Näkäinen funktio syö argumenttinsa palauttaen uuden funktion, joka on valmis vastaanottamaan uuden argumentin. Virrat (stream) ovat nälkäisten funktioiden tyyppi, joka voi kuluttaa määrittelemättömän määrän `unit` argumentteja palauttaen aina uuden numero/virta -parin:

```
Stream = μA. Unit → {Nat,A};
```

Kuten listoille voidaan virroille määritellä `hd` ja `tl` funktiot:

```
    hd = λs:Stream. (s unit).1;
▶   hd : Stream → Nat

    tl = λs:Stream. (s unit).2;
▶   tl : Stream → Stream
```

Esimerkkinä virran käytöstä, luodaan `fix`-funktiota avuksi käyttäen virta $\{0,1,2,\ldots\}$, ja tarkastellaan sen käyttäytymistä funktioiden `hd` ja `tl` avulla:

```
    upfrom0 = fix (λf: Nat → Stream. λn:Nat. λ_:Unit {n, f (succ n)}) 0;
▶   upfrom0 : Stream

    hd upfrom0
▶   0
    hd (tl (tl (tl upfrom0)));
▶   3 : Nat
```

Virtoja voi edelleen yleistää prosesseiksi, jotka saavat syötteen (luonnollisen luvun) ja palauttavat uuden luvun sekä uuden prosessin.

```
Process = μA. Nat→{Nat,A};
```

Interaktio prosessien kanssa hoidetaan funtioilla `curr` ja `send`. Näistä ensimmäinen palauttaa prosessin nykyarvon ja jälkimmäinen lähettää arvon prosessille.[1]

[1] Kirjan esimerkin `curr` antaa myös syötteen prosessille. Syötteenä annetaan nolla (0), jolla ei ole vaikutusta kirjan summaa laskevaan funktioon. Esimerkiksi keskiarvoa laskevan prosessin tilaa ei `curr`-funktiolla

```
    curr = λs:Process. (s 0).1;
▶   curr : Process → Nat

    send = λn:Nat. λs:Process. (s n).2
▶   send : Nat → Process → Process
```

Esimerkkinä prosessista on funktio `sum`, joka laskee syötteenään saamiensa numeroiden summaa.

```
    sum = fix (λf: Nat→Process. λacc:Nat λn:Nat.
                let newacc = plus acc n in
                {newacc, f newacc}) 0;
▶   curr : Process

    curr (send 20 (send 3 (send 5 sum)));
▶   28 : Nat
```

2.3 Oliot

Rekursiivisten tyyppien avulla on myös mahdollista määritellä olioita. Esimerkiksi `Counter` on tyyppi, joka tarjoaa aksessorit sisältämänsä numeron kysymiseen ja kasvattamiseen. Näin määritellyt tyypit ovat varsinaisista oliokielistä poiketen kuitenkin puhtaan funktionaalisia: olion sisäistä tilaa muuttavat aksessorikutsut palauttavat uuden olion, eivätkä varsinaisesti muokkaa olion sisäistä tilaa. Erona prosesseihin on se, että olio on rekursiivisesti määritelty *tietue* (record), joka sisältää funktioita, kun taas prosessi on rekursiivisesti määritelty *funktio*, joka palauttaa monikon (tuple). Näkökulman vaihdon suurin etu on siinä, että tietue voi sisältää useampia kuin yhden operaation. Alla on esimerkki `Counter`-tyypistä ja sen käytöstä. Esimerkin `Counter` sisältää yllä mainittujen kysely- ja lisäysaksessorien lisäksi myös arvon vähennys metodin (`dec`).

```
    Counter = μC. {get:Nat, inc:Unit→C, dec:Unit→C};

    c = let create = fix (λf: {x:Nat}→Counter. λs: x:Nat.
                            {get = s.x,
                             inc = λ_:Unit. f {x=succ(s.x)},
                             dec = λ_:Unit. f {x=pred(s.x)} })
       in create {x=0};

▶   c : Counter

    c1 = c.inc unit;
    c2 = c1.inc unit;
    c2.get;
▶   2 : Nat
```

tulisi mennä kyselemään.

2.4 Tyypitetty `fix`

Rekursiivisten tyyppien avulla on mahdollista määritellä hyvin tyypitetty versio `fix`-funktiosta mille tahansa tyyppiä `T` käsitteleville funktioille. Tämä määritelmä (alla) on tyyppimääritykset poistamalla täysin sama kuin kirjan sivulla 65 annettu määritelmä `fix`:lle.

```
    fix_T = λf:T→T. (λx:(μA.A→T). f (x x)) (λx:(μA.A→T). f (x x));
▶   fix_T : (T→T) → T
```

Tässä määritelmässä rekursiivista tyyppiä käytetään alilauseiden `x x` tyypittämiseen. Kyseisen termin tyypittäminen vaatii nuoli-tyypin, jonka ala (domain) on `x`:n itsensä tyyppi.

Mahdollisuus määritellä rekursiivisia tyyppejä johtaa vahvan normalisaatio-ominaisuuden (strong normalization property) rikkoutumiseen: fix_T-funktiolla on mahdollista määritellä hyvin tyypitetty termi, jonka evaluaatio hajautuu, kun sitä sovelletaan `unit`-arvolle.

```
    diverge_T = λ_:Unit. fix_T (λx:T. x);
▶   diverge_T : Unit → T
```

3 Muodollisuudet

Kirjallisuudesta löytyy kaksi peruslähestymistapaa rekursiivisiin tyyppeihin. Nämä ratkaisut ratkaisut eroavat kysymyksen ”mikä on tyypin `μX.T` ja sen yhden askeleen verran lavennetun version suhde?” suhteen. Esimerkiksi, mikä on `NatList`:n ja `<nil:Unit, cons{Nat, NatList}>` suhde?

Yhtäläisrekursiivinen (equi-recursive) lähestymistapa samaistaa nämä lauseet määritelmällisesti samoiksi. Koska ne kuvaavat saman äärettömän puun, ovat ne kovattavissa toisillaan missä tahansa kontekstissa. Tyypintarkastajan vastuulle jää sen varmistaminen, että yhdentyyppistä versiota termistä odottava funktio hyväksyy myös toisen muotoisen termin.

Hyvä puoli tässä lähestymistavassa on se, ettei sen esitystapa eroa aiemmin esitetystä deklaratiivisesta esitystavasta, muuten kuin sen suhteen, että tyyppimäärittelyt voivat olla äärettömiä (säännöllisiä). Aikaisemman määritelmät, turvallisuusteoreemat ja todistukset käyvät sellaisenaan, olettaen etteivät ne riipu induktiosta tyyppilausekkeiden suhteen. Käytännössä tämä lähestymistapa on kuitenkin työläs toteuttaa, sillä tyypintarkastusalgoritmit eivät pysty suoraan käsittelemään äärettömiä tyyppejä.

Isorekursiivinen lähestymistapa käsittelee lavennettua ja laventamatonta määritelmää erillisinä, mutta isomorfisina. Rekursiivisen tyypin `μX.T` laventaminen tapahtuu korvaamalla `X` kaikki esiitymät vartalossa `T`: `[X ↦ (μX.T)]T`. Isorekursiivisessa systeemissa jokaista tyyppiä `μX.T` vastaa kaksi kielen tarjoamaa primitiivifunktiota:

```
unfold[μX.T]  :  μX.T → [X ↦ μX.T]T
fold[μX.T]    :  [X ↦ μX.T]T → μX.T
```

Nämä ovat määritelty kuvan 1 mukaisesti.

Tämä lähestymistapa on helpompi implementoida, mutta vaatii ohjelmoijalta enemmän työtä: `fold` ja `unfold` ohjeiden antamisen aina rekursiivisia tyyppejä käsiteltäessä. Käytännössä nämä kutsut voidaan peittää muilla käskyillä. Esimerkiksi Javassa luokkamäärittely luo rekursiivisen tyypin ja olion metodikutsu peittää `unfold`:n. Alla on esimerkki `NatList`-tyypin isorekursiivisesta määrittelystä ja käytöstä.

```
NLBody = <nil:Unit, cons:{Nat, NatList}>;

nil = fold [NatList] (<nil=unit> as NLBody);

cons = λn:Nat. λl:NatList. fold [NatList] <cons={n, l}> as NLBody;

isnil = λl:NatList.
          case unfold [NatList] l of
            <nil=u> ⇒ true
          | <cons=p> ⇒ false;

hd = λl:NatList.
          case unfold [NatList] l of
            <nil=u> ⇒ 0
          | <cons=p> ⇒ p.1;

tl = λl:NatList.
          case unfold [NatList] l of
            <nil=u> ⇒ l
          | <cons=p> ⇒ p.2;
```

4 Rekursiivisten tyyppien alityypitys

Jos tyyppi `Even` on tyypin `Nat` alityyppi, niin mikä on rekursiivisten tyyppien μ`X.Nat`$\rightarrow$`(Even`$\times$`X)` ja μ`X.Even`$\rightarrow$`(Nat`$\times$`X)` suhde?

Intuitio rekursiivisten tyyppien alityypityksessä on ajatella näitä yksinkertaisina prosesseina, jotka saatuaan numeerisen syötteen palauttavat uuden numeron ja prosessin, joka on valmis vastaanottamaan uuden syötteen. Tyyppiä μ`X.Nat`$\rightarrow$`(Even`$\times$`X)` oleva prosessi hyväksyy minkä tahansa luonnollisen luvun, palauttaen kokonaisluvun. Jälkimmäisen tyypin, μ`X.Even`$\rightarrow$`(Nat`$\times$`X)`, vaatimukset ovat puolestaan selkeästi vähäisemmät: tätä tyyppiä olevat prosessit hyväksyvät syötteeksi vain kokonaislukuja ja voivat puolestaan palauttaa mitä tahansa lukuja. Koska ensimmäiselle tyypille asetetaan kovemmat vaatimukset, katsotaan sen olevan jälkimmäisen alityyppi — ts. alityypin lähtöjoukko on laajempi, kuin ylityypin ja tulojoukko suppeampi. Kuva 2 havainnollistaa rekursiivisten tyyppien periytymistä.

t ::=	...	*termit:*
	fold [T] t	*supistaminen*[2]
	unfold [T] t	*laventaminen*[3]
v ::=	...	*arvot:*
	fold [T] v	*supistaminen*
T ::=	...	*tyypit:*
	X	*tyyppimuuttuja*
	μX.T	*rekursiivinen tyyppi*

Uudet evaluointisäännöt $\boxed{t \to t'}$

$$\texttt{unfold [S] (fold [T] } v_1\texttt{)} \to v_1 \qquad \text{(E-UnfldFld)}$$

$$\frac{t_1 \to t_1'}{\texttt{fold [T] } t_1 \to \texttt{fold [T] } t_1'} \qquad \text{(E-Fld)}$$

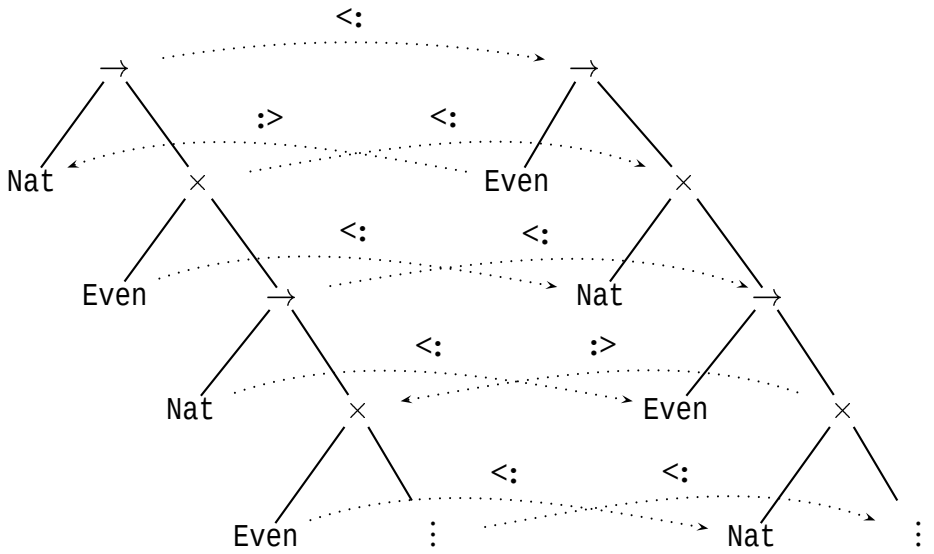
$$\frac{t_1 \to t_1'}{\texttt{unfold [T] } t_1 \to \texttt{unfold [T] } t_1'} \qquad \text{(E-Unfld)}$$

Uudet tyypityssäännöt $\boxed{\Gamma \vdash t : T}$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash \texttt{fold [U] } t_1 : U} \qquad \text{(T-Fld)}$$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash \texttt{unfold [U] } t_1 : [X \mapsto U]T_1} \qquad \text{(T-Unfld)}$$

Kuva 1: Isorekursiivisen lähestymistavan vaatimat säännöt.



Kuva 2: Rekursiivisten tyyppien μX.Nat $\rightarrow$ (Even$\times$X) ja μX.Even $\rightarrow$ (Nat$\times$X) alityypitystä havainnollistava kaavio.

4.1 Yhtäläisrekursiivinen lähestymistapa

Koska isorekurisiivisen lähestymistavan toteutus on suoraviivaista, keskitytään tässä yhteydessä yhtäläisrekursiivisen lähestymistavan tyypintarkastajien teoreettisiin perusteisiin. Rekursiivisten tyyppien alityypitys on intuitiivisesti äärettömien tyyppien alityypityksen derivaationa. Matemaattisen täsmällinen määritelmä alityypitykselle saadaan käyttämällä *koinduktiota*.

Määritelmä: Funktio $F \in \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ on *monotoninen*, jos $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$, jossa $\mathcal{U}$ tarkoittaa universaalia joukkoa, joka sisältää ”kaiken maailmassa” ja $\mathcal{P}(\mathcal{U})$ tarkoittaa kaikkien $\mathcal{U}$:n alijoukkojen joukkoa.

Määritelmä: Olkoon X $\mathcal{U}$:n alijoukko.

1. X on F*-suljettu*, jos $F(X) \subseteq X$
2. X on F*-konsistentti*, jos $X \subseteq F(X)$
3. X on F:n *kiintopiste*, jos $X = F(X)$

Intuitiivisesti $\mathcal{U}$:n voi ajatella sisältävän väitteitä ja F:n olevan todistus relaatio, joka kertoo, mitä uusia päätelmiä näistä väitteistä (premisseistä) voidaan johtaa. F-suljettu joukko on tällöin sellainen joukko, jota ei voi kasvattaa lisäämällä uusia F:n tunnistamia alkioita, koska se sisältää jo kaikki johdettavissa olevat päätelmät. F-konsistentti joukossa puolestaan jokaista väitettä tukee muut joukon sisältämät väitteet, se on siis itsensä perusteleva joukko. F:n kiintopiste on sekä suljettu että konsistentti, sisältäen kaikkien jäsentensä vaatimat premissit sekä kaikki pääteltävissä olevat johtopäätökset, eikä mitään muuta.

Esimerkiksi funktio E_1 on määritelty seuraavasti kolmielementtisessä avaruudessa $\mathcal{U} = a, b, c$:

$$\begin{array}{ll} E_1(\varnothing) = \{c\} & E_1(\{a,b\}) = \{c\} \\ E_1(\{a\}) = \{c\} & E_1(\{a,c\}) = \{c\} \\ E_1(\{b\}) = \{c\} & E_1(\{b,c\}) = \{a,b,c\} \\ E_1(\{c\}) = \{b,c\} & E_1(\{a,b,c\}) = \{a,b,c\} \end{array}$$

Avaruus $\mathcal{U}$ sisältää tällöin yhden E_1-suljetun joukon ($\{a,b,c\}$), neljä E_1-konsistenttia joukkoa ($\varnothing$, $\{c\}$, $\{b,c\}$ ja $\{a,b,c\}$) sekä yhden kiintopisteen ($\{a,b,c\}$).

Knaster-Tarskin teoreema sanoo, että

1. Kaikkien F-suljettujen joukkojen leikkaus on pienin F:n kiintopiste. Tätä joukkoa, joka on myös pienin F-suljettu joukko, merkitään μF:llä.

2. Kaikkien F-konsistenttien joukkojen unioni on suurin F:n kiintopiste. Tätä joukkoa, joka on myös suurin F-konsistentti joukko, merkitään νF:llä.

Näistä todistetaan vain jälkimmäinen, ensimmäisen todistuksen ollessa symmetrinen. Olkoon $C = \{X | X \subseteq F(X)\}$ kaikkien F-konsistenttien joukkojen kokoelma ja P näiden unioni. Koska F on monotoninen ja kaikille $X \in C$ tiedetään, että X on F-konsistentti ja, että $X \subseteq C$, on $C \subseteq F(X) \subseteq F(P)$. Näin ollen $P = \cup_{X \in C} X \subseteq F(P)$, eli P on F-konsistentti. Määritelmänsä mukaisesti P on tällöin suurin F-konsistentti joukko. Tämän jälkeen tulee todistaa myös, että P on myös F-suljettu. Koska F on monotoninen saadaan $F(P) \subseteq F(F(P))$. C:n määritelmän mukaisesti tämjä tarkoittaa, että $F(P) \in C$. Täten mille tahansa C:n jäsenelle $F(P) \subseteq P$, eli P on F-suljettu. Koska P on suurin F-konsistentti joukko ja P on myös F:n kiintopiste, on P suurin kiintopiste.

Knaster-Tarskin teoreemasta seuraa, että

1. *Induktioperiaate*: jos X on F-suljettu, niin $\mu F \subseteq X$.

2. *Koinduktioperiaate*: jos X on F-konsistentti, niin $X \subseteq \nu F$.

Intuitio näiden määritelmien takana seuraa siitä, että joukko X mielletään predikaatiksi, joka esitetään sen piirteiden joukkona, sinä $\mathcal{U}$:n osajoukkona, jolla predikaatti on totta. Sen osoittaminen, että X on totta elementille x on sama kuin sen osoittaminen, että x kuuluu joukkoon X. Induktioperiaatteen mukaisesti mikä tahansa ominaisuus, jonka piirteiden joukko on suljettu F:n alla, eli F säilyttää kyseisen piirteen, on totta myös induktiivisesti määritellylle joukolle μF.

Koinduktioperiaate puolestaan tarjoaa välineet sen todistamiselle, että elementti x kuuluu koinduktiivisesti määriteltyyn joukkoon νF. Tämän osoittamiseksi riittää löytää jokin F-konsistentti joukko X, jolla $x \in X$.

5 Äärettömät ja äärelliset tyypit

Tyypejä on mahdollista lähestyä äärellisinä tai äärettöminä puina. Tässä yhteydessä riittää kolmen tyyppikonstruktorin käsitteleminen: $\rightarrow$, $\times$ ja $\mathtt{Top}$. Tyyppejä käsitellään (mahdollisesti äärettöminä) puina, joiden solmuja merkataan jollain näistä symboleista. Ykkösistä ja kakkosista koostuvien sekvenssien joukkoa merkataan {1,2}*, tyhjää sekvenssiä merkillä $\bullet$ ja i^k tarkoittaa k:ta i:n kopiota. Jos π ja σ ovat sekvenssejä, tarkoittaa π,σ näiden katenaatiota.

Puutyyppi eli puu, on osittaisfunktio $\mathtt{T} \in \{1,2\}* \rightharpoonup \{\rightarrow, \times, \mathtt{Top}\}$, joka täyttää ehdot:

- $\mathtt{T}(\bullet)$ on määritelty;
- jos $\mathtt{T}(\pi,\sigma)$ on määritelty niin $\mathtt{T}(\pi)$ on määritelty;
- jos $\mathtt{T}(\pi) = \rightarrow$ tai $\mathtt{T}(\pi) = \times$, niin $\mathtt{T}(\pi,1)$ ja $\mathtt{T}(\pi,2)$ ovat määritelty;
- jos $\mathtt{T}(\pi) = \mathtt{Top}$ niin $\mathtt{T}(\pi,1)$ ja $\mathtt{T}(\pi,2)$ ovat määrittelemättömät;

Puutyyppi $\mathtt{T}$ on äärellinen, jos *dom*($\mathtt{T}$) on äärellinen. Kaikkien puutyyppien joukkoa merkitään symbolilla $\mathcal{T}$ ja kaikkien äärellisten puiden joukkoa symbolilla $\mathcal{T}_f$.

Notaationaalisista mukavuussyista puulle $\mathtt{T}$ merkitään $\mathtt{T}(\bullet) = \mathtt{Top}$. Jos $\mathtt{T}_1$ ja $\mathtt{T}_2$ ovat puita, tarkoittaa

$\mathtt{T}_1 \times \mathtt{T}_2$
: puuta, jolla $(\mathtt{T}_1 \times \mathtt{T}_2)(\bullet) = \times$ ja $(\mathtt{T}_1 \times \mathtt{T}_2)(i,\pi) = \mathtt{T}_i(\pi)$ ja

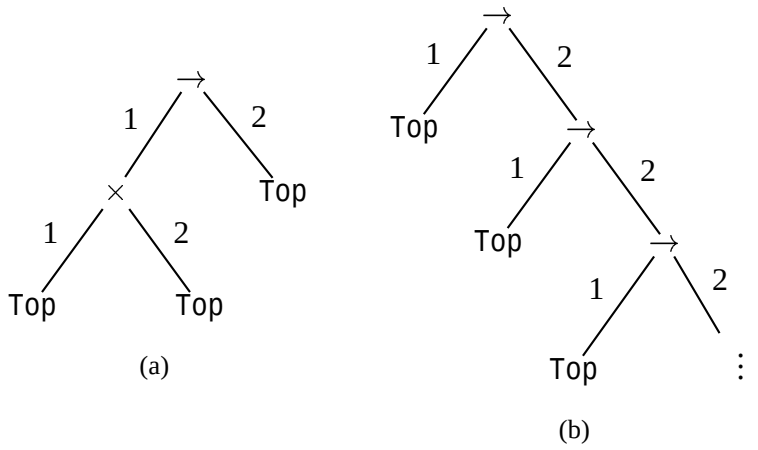
$\mathtt{T}_1 \rightarrow \mathtt{T}_2$
: puuta, jolla $(\mathtt{T}_1 \rightarrow \mathtt{T}_2)(\bullet) = \rightarrow$ ja $(\mathtt{T}_1 \rightarrow \mathtt{T}_2)(i,\pi) = \mathtt{T}_i(\pi)$

missä $i = 1,2$.

Kuvan 3(a) äärellisessä puutyypissä $(\mathtt{Top} \times \mathtt{Top}) \rightarrow \mathtt{Top}$

- $\mathtt{T}(\bullet) = \rightarrow$,
- $(\mathtt{T}(1) = \times$ ja
- $\mathtt{T}(2) = \mathtt{T}(1,1) = \mathtt{T}(1,2) = \mathtt{Top}$.

Kuvan 3(b) ääretön puu, $\mathtt{Top} \rightarrow (\mathtt{Top} \rightarrow (\mathtt{Top} \rightarrow \ldots))$, vastaa tyyppiä $\mathtt{T}$, jonka määritelmä on $\mathtt{T}(2^k) = \rightarrow$ ja $\mathtt{T}(2^k,1) = \mathtt{Top}$ kaikilla $k \geq 0$.



Kuva 3: Vasemmalla äärellinen puutyyppi $(\mathsf{Top} \times \mathsf{Top}) \to \mathsf{Top}$ ja oikealla ääretön puutyyppi $\mathsf{Top} \to (\mathsf{Top} \to (\mathsf{Top} \to \ldots))$.

6 Alityypityksen teoria

Äärellisten puiden alityypitysrelaatio määritellään pienimmäksi määritellyn avaruuden monotoonisten funktioiden kiintopisteeksi ja yleisten puiden osalta suurimmaksi kiintopisteeksi. Äärellisten puiden tyypityksen osalta tämä avaruus on joukko $\mathcal{T}_f \times \mathcal{T}_f$ äärellisten puiden pareja. Alityypityksen generoiva funktio on kuvaus tämän avaruuden alijoukkojen välillä ja niiden kiintopiste relaatio $\mathcal{T}_f$:ssä. Yleisesti puiden tyypityksen osalta avaruus määritellään pariksi $\mathcal{T} \times \mathcal{T}$.

6.1 Äärellisten puiden alityypitys

Kaksi äärellistä puuta S ja T ovat alityyppisuhteessa (”S on T:n alityyppi”), jos $(\mathsf{S},\mathsf{T}) \in \mu S_f$, jossa $S_f \in \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f) \to \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f)$ on määritelty seuraavasti:

$$\begin{array}{rcl} S_f(R) & = & \{(\mathsf{T},\mathsf{Top}) \mid \mathsf{T} \in \mathcal{T}_f\} \\ & \cup & \{(\mathsf{S}_1 \times \mathsf{S}_2, \mathsf{T}_1 \times \mathsf{T}_2) | (\mathsf{S}_1,\mathsf{T}_1),(\mathsf{S}_2,\mathsf{T}_2) \in R\} \\ & \cup & \{(\mathsf{S}_1 \to \mathsf{S}_2, \mathsf{T}_1 \to \mathsf{T}_2) | (\mathsf{T}_1,\mathsf{S}_1),(\mathsf{S}_2,\mathsf{T}_2) \in R\}. \end{array}$$

Tämä funktio toteuttaa aiemmin esitetyn alityypityksen määritelmän näiden päättelysääntöjen nojalla:

$$\frac{}{\mathsf{T} <: \mathsf{Top}}$$

$$\frac{\mathsf{S}_1 <: \mathsf{T}_1 \quad \mathsf{S}_2 <: \mathsf{T}_2}{\mathsf{S}_1 \times \mathsf{S}_2 <: \mathsf{T}_1 \times \mathsf{T}_2}$$

$$\frac{\mathsf{T}_1 <: \mathsf{S}_1 \quad \mathsf{S}_2 <: \mathsf{T}_2}{\mathsf{S}_1 \to \mathsf{S}_2 <: \mathsf{T}_1 \to \mathsf{T}_2}$$

Näissä päättelysäännöissä `S <: T`:n esiintymät viivan yläpuolella luetaan ”jos pari `(S,T)` on syötteessä S_f:lle” ja viivan alapuolella ”niin `(S,T)` on tuloksessa”.

6.2 Äärettömien puiden alityypitys

Yleisesti puiden, äärettöminen ja äärellisten, alityypityksen generoiva funktio S määritellään vastaavalla tavalla, kuin äärellisten puiden alityypitys (luku 6.1), sillä erolla, että $\mathcal{T}_f$ korvataan $\mathcal{T}$:llä.

Myös äärettömien puiden alityypitysrelaatio on transitiivinen. Tämä on määritelty seuraavasti: Relaatio $R \subseteq \mathcal{U} \times \mathcal{U}$ on *transitiivinen*, jos R on suljettu monotonisen funktion $TR(R) = \{(x,y) | \exists z \in \mathcal{U}.(x,z),(z,y) \in R\}$ suhteen eli, jos $TR(R) \subseteq R$.

Olkoon $F \in \mathcal{P}(\mathcal{U} \times \mathcal{U}) \to \mathcal{P}(\mathcal{U} \times \mathcal{U})$ on monotoninen funktio. Jos $TR(F(R)) \subseteq F(TR(R))$ millä tahansa $R \subseteq \mathcal{U} \times \mathcal{U}$, on νF transitiivinen.

Koska νF on kiintopiste ($\nu F = F(\nu F)$), seuraa siitä, että $TR(\nu F) = TR(F(\nu F))$. Toisin sanoen $TR(\nu F)$ on F-konsistentti, ja koinduktioperiaatteen mukaisesti $TR(\nu F) \subseteq \nu F$. Samoin νF on transitiivinen transitiivisuuden määritelmän nojalla.