



## **Paketti III: SOVELLUKSIA**

1. Teoreemantodistin OTTER
2. Logiikkaohjelmointi
3. Ohjelmien oikeellisuustarkastelut



## **1 Teoreemantodistin OTTER**

- Syötetiedoston sisältö
- Käyttöohjeita
- Tulosteiden tulkitseminen
- Kotitehtävän palautus
- Saavutuksia matematiikassa



## 1.1 Syötetiedoston sisältö

- OTTER noudattaa lauseiden osalta seuraavaa syntaksia:

$(\forall x(\forall y(\forall z\varphi)))$	$\rightsquigarrow$	<code>(all x y z <math>\varphi</math>)</code>
$(\exists x(\exists y(\exists z\varphi)))$	$\rightsquigarrow$	<code>(exists x y z <math>\varphi</math>)</code>
$(\varphi \wedge \psi)$	$\rightsquigarrow$	<code>(<math>\varphi</math> &amp; <math>\psi</math>)</code>
$(\varphi \vee \psi)$	$\rightsquigarrow$	<code>(<math>\varphi</math>   <math>\psi</math>)</code>
$(\varphi \rightarrow \psi)$	$\rightsquigarrow$	<code>(<math>\varphi</math> -&gt; <math>\psi</math>)</code>
$(\varphi \leftrightarrow \psi)$	$\rightsquigarrow$	<code>(<math>\varphi</math> &lt;-&gt; <math>\psi</math>)</code>
$(t_1 = t_2)$	$\rightsquigarrow$	<code>(<math>t_1 = t_2</math>)</code>
$(\neg\varphi)$	$\rightsquigarrow$	<code>(-<math>\varphi</math>)</code>



### Esimerkki.

$\forall x\forall y\exists zP(x,y,z)$   $\rightsquigarrow$  `all x y (exists z P(x,y,z)).`

$\forall x(P(x) \wedge (Q(x) \vee R(x)))$   $\rightsquigarrow$  `all x (P(x) & (Q(x) | R(x))).`

- Eri symbolit erotetaan toisistaan sijainnin perusteella. OTTER sallii samannimiset vakio-, muuttuja-, funktio- ja predikaattisymbolit!
- OTTER erottelee muuttujat vakioista kvanttorien perusteella (näin mahdolliset "vapaat" muuttujaesiintymät tulkitaan vakioiksi).
- OTTER muuntaa lauseet automaattisesti klausuuleiksi, joita voi käyttää myös tiettyjen syntaksivirheiden tunnistamiseen.

**Esimerkki.** Lausekkeessa `(all x y(P(x,y)))` esiintyvä `y` on sekä predikaatti- että vakiosymboli ja `P` funktiosymboli, mutta `y` tulkitaan muuttujaksi ja `P` predikaatiksi lausekkeessa `(all x y (P(x,y)))`.



### OTTERille laadittava syötetiedosto

- Yksinkertainen esimerkki syötetiedosta on annettu alla.
- `set(auto)`-asetuksella OTTER valitsee itse käytettävät päättelysäännöt (mm. resoluutiosääntö muunnelmiseen).

---

```
set(auto).  
formula_list(usable).  
  
% tähän tulee sitten tarvittava lausejoukko  
  
% ja kyselyn negaatio (vain yksi kysely kerrallaan)  
  
end_of_list.
```

---

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



### Esimerkki. Palautetaan mieliin aikaisempi tartuntavaaraesimerkki.

```
set(auto).  
formula_list(usable).  
% Section A: database  
tapaa(a,b). tapaa(a,c). sairastaa(c).  
  
% Section B: definitions  
all x y (tapaa(x,y) -> tapaa(y,x)).  
all x y (tapaa(x,y) & sairastaa(y) -> tartuntavaarassa(x)).  
all x y (tapaa(x,y) & tartuntavaarassa(y) -> tartuntavaarassa(x)).  
  
% Section C: negation of the query  
-(tartuntavaarassa(a) & tartuntavaarassa(b)).  
  
end_of_list.
```

- OTTER pystyy osoittamaan lausejoukon helposti ristiriitaiseksi.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



## 1.2 Käyttöohjeita

- OTTER otetaan käyttöön Atk-keskuksen unix-koneissa komennolla  
`use otter`
- Manuaaliin löytyy linkki kotitehtäväpalvelimesta sivulta  
`http://logic.tcs.hut.fi/~ltp/`
- OTTERin voi käynnistää esim. seuraavilla tavoilla:  
`otter < file.in`  
`otter < file.in > file.out`  
`otter < file.in | less`
- Hakemistosta `/p/edu/tik-79.144` löytyy myös dokumentaatiota ja esimerkkejä (pääasiassa matematiikasta).

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



## 1.3 Tulosteiden tulkitseminen

OTTERin tulostiedostosta löytyvät mm.

- Lauseille automaattisesti haetut klausuulimuodot.
- Klausuulien luokittelu (tukijoukon identifiointi).
- Annetuista klausuuleista johdetut uudet klausuulit, kun OTTER yrittää johtaa tyhjän klausuulin.
- Mahdollisesti löydetty todistus, johon eristetään tyhjän klausuulin johtamisessa tarvittavat klausuulit.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



**Esimerkki.** Tutustutaan tarkemmin OTTERin tulosteisiin tartuntavaaraesimerkissä.

1. Klausuulimuodosta tulee seuraava:

```
list(usable).
0 [] tapaa(a,b).
0 [] tapaa(a,c).
0 [] sairastaa(c).
0 [] -tapaa(x,y)|tapaa(y,x).
0 [] -tapaa(x,y)| -sairastaa(y)|tartuntavaarassa(x).
0 [] -tapaa(x,y)| -tartuntavaarassa(y)|tartuntavaarassa(x).
0 [] -tartuntavaarassa(a)| -tartuntavaarassa(b).
end_of_list.
```

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



2. set(auto)-asetuksen myötä OTTER päättää itse tukijoukkoon (engl. set of support) tulevat klausuulit:

```
-----> process usable:
** KEPT (pick-wt=6): 1 [] -tapaa(x,y)|tapaa(y,x).
** KEPT (pick-wt=7): 2 [] -tapaa(x,y)| -sairastaa(y)|
                        tartuntavaarassa(x).
** KEPT (pick-wt=7): 3 [] -tapaa(x,y)| -tartuntavaarassa(y)|
                        tartuntavaarassa(x).
** KEPT (pick-wt=4): 4 [] -tartuntavaarassa(a)|
                        -tartuntavaarassa(b).

-----> process sos:
** KEPT (pick-wt=3): 5 [] tapaa(a,b).
** KEPT (pick-wt=3): 6 [] tapaa(a,c).
** KEPT (pick-wt=2): 7 [] sairastaa(c).
```

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



3. Tämän jälkeen yritetään johtaa tyhjä klausuuli, mikä onnistuukin:

===== start of search =====

```
given clause #1: (wt=3) 5 [] tapaa(a,b).
given clause #2: (wt=2) 7 [] sairastaa(c).
given clause #3: (wt=3) 6 [] tapaa(a,c).
given clause #4: (wt=2) 9 [hyper,6,2,7] tartuntavaarassa(a).
given clause #5: (wt=3) 8 [hyper,5,1] tapaa(b,a).
given clause #6: (wt=3) 10 [hyper,6,1] tapaa(c,a).
given clause #7: (wt=2) 11 [hyper,8,3,9] tartuntavaarassa(b).

-----> EMPTY CLAUSE at 0.00 sec -----> 13 [hyper,11,4,9] $F.
```

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



4. Tämän jälkeen OTTER eristää johdon tyhjälle klausuulille:

Length of proof is 3. Level of proof is 2.

----- PROOF -----

```
1 [] -tapaa(x,y) | tapaa(y,x).
2 [] -tapaa(x,y) | -sairastaa(y) | tartuntavaarassa(x).
3 [] -tapaa(x,y) | -tartuntavaarassa(y) | tartuntavaarassa(x).
4 [] -tartuntavaarassa(a) | -tartuntavaarassa(b).
5 [] tapaa(a,b).
6 [] tapaa(a,c).
7 [] sairastaa(c).
8 [hyper,5,1] tapaa(b,a).
9 [hyper,6,2,7] tartuntavaarassa(a).
11 [hyper,8,3,9] tartuntavaarassa(b).
13 [hyper,11,4,9] $F.

----- end of proof -----
```

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



### Huomioita.

- Mikäli OTTER ei löydä todistusta, OTTER voi pysähtyä ilmoittaen "Search stopped because sos empty." tai jäädä ikuisen silmukkaan.
- Muuttujasidontojen eristämistä varten OTTERissa voi määritellä \$ans-alkuisia predikatteja, joiden argumenteiksi kirjataan mielenkiinnon kohteena olevat muuttujat.

**Esimerkki.** Haetaan jokin tartuntavaarassa oleva henkilö kyselyllä  $(\text{exists } x (\text{tartuntavaarassa}(x) \ \& \ \$\text{ans}(x)))$ :

```
2 [] -tapaa(x,y) | -sairastaa(y) | tartuntavaarassa(x) .
4 [] -tartuntavaarassa(x) | -$$ans(x) .
6 [] tapaa(a,c) .
7 [] sairastaa(c) .
9 [hyper,6,2,7] tartuntavaarassa(a) .
10 [binary,9.1,4.1] -$$ans(a) .
```



## 1.4 Kotitehtävän palautus

- Ratkaisuksi laaditaan 3 syötetiedostoa OTTERille.
- Kohtaan 3a palautetaan syötetiedosto, joka sisältää pelkät predikaattien määritelmät (ei tietokantaa eikä kyselyä).
- Kohtiin 3b ja 3c palautetaan syötetiedosto, joka sisältää em. määritelmien lisäksi myös tietokannan ja tähän liittyvän kyselyn.
- Kohdan 3a ratkaisuksi palautettua syötetiedostoa käytetään
  1. kohdassa 3a määritelmien konsistenssin tarkastamiseen,
  2. kohdassa 3d vertailuun mallimääritelmien kanssa ja
  3. kohdassa 3e testikyselyn evaluointiin jonkin tietokannan suhteen.



## 1.5 Saavutuksia matematiikassa

- OTTERia on käytetty mm. matematiikkaan liittyvien avointen ongelmien ratkomiseen.
- Ensimmäisenä merkittävänä ongelmana pystyttiin osoittamaan kahden Boolean algebroille esitetyn aksiomatisoinnin ekvivalenssi (Huntingtonin ja Robbinsin määritelmät).
- Tuloksen yksityiskohtia on selvitetty tarkemmin sivulla  
<http://www-unix.mcs.anl.gov/~mccune/papers/robbins/>
- Muita OTTERilla (ja OTTERiin liittyvillä ohjelmilla) osoitettuja tuloksia on raportoitu sivulla  
[http://www-unix.mcs.anl.gov/AR/new\\_results/](http://www-unix.mcs.anl.gov/AR/new_results/)

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



## 2 Logiikkaohjelmointi

- Logiikkaohjelmien syntaksi
- Herbrand-malleihin perustuva semantiikka
- Resoluutio logiikkaohjelmien tapauksessa
- PROLOGin hakumeکانismi

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio





## 2.1 Logiikkaohjelmien syntaksi

Logiikkaohjelmointi (esim. PROLOG) perustuu klausuulien aliluokkaan.

**Määritelmä.** Mahdollisesti muuttujia sisältävää klausuulia

$C = \{A_1(\vec{t}_1), \dots, A_n(\vec{t}_n), \neg B_1(\vec{u}_1), \dots, \neg B_m(\vec{u}_m)\}$  kutsutaan

1. *Horn-klausuuliksi*, jos  $n \leq 1$ ,
2. *ohjelmaklausuuliksi*, jos  $n = 1$ , ja
3. *maaliklausuuliksi*, jos  $n = 0$ .

**Esimerkki.** Klausuulit  $\{N(0)\}$  ja  $\{N(s(x)), \neg N(x)\}$  ovat ohjelmaklausuuleja, kun taas  $\{\neg N(s(0)), \neg N(s(s(x)))\}$  on maaliklausuuli. Näistä jokainen on Horn-klausuuli.

**Määritelmä.** Logiikkaohjelma  $P$  on joukko ohjelmaklausuuleja.



## Logiikkaohjelmien suhde predikaattilogiikkaan

Klausuulimuotojen ja normaalimuotojen välisten yhteyksien perusteella:

1. Ohjelmaklausuuli  $C = \{A(\vec{t}), \neg B_1(\vec{u}_1), \dots, \neg B_m(\vec{u}_m)\}$  vastaa universaalisti kvantifioitua lausetta

$$\forall x_1 \dots \forall x_n (B_1(\vec{u}_1) \wedge \dots \wedge B_m(\vec{u}_m) \rightarrow A(\vec{t}))$$

missä  $x_1, \dots, x_n$  ovat termijonoissa  $\vec{u}_1, \dots, \vec{u}_m$  ja  $\vec{t}$  mahdollisesti esiintyvät muuttujat.

2. Maaliklausuuli  $G = \{\neg B_1(\vec{u}_1), \dots, \neg B_m(\vec{u}_m)\}$  vastaa eksistentiaalisesti kvantifioidun lauseen

$$\phi_G = \exists x_1 \dots \exists x_n (B_1(\vec{u}_1) \wedge \dots \wedge B_m(\vec{u}_m))$$

**negaatiota**  $\neg \phi_G$ , missä  $x_1, \dots, x_n$  ovat termijonoissa  $\vec{u}_1, \dots, \vec{u}_m$  mahdollisesti esiintyvät muuttujat.



$\implies$  Ohjelmaklausuulien muuttujat ovat universaalisti kvantifioidut ja maaliklausuulin muuttujat ovat eksistensiaalisesti kvantifioidut.

**Esimerkki.** Tarkastellaan logiikkaohjelmaa

$$P = \{\{N(0)\}, \{N(s(x)), \neg N(x)\}\}$$

ja maaliklausuulia (kyselyn negaatio)  $G = \{\neg N(s(0)), \neg S(s(s(x)))\}$ .

- Ohjelma  $P$  vastaa lausejoukkoa  $\Sigma = \{N(0), \forall x(N(x) \rightarrow N(s(x)))\}$ .
- Maaliklausuuli  $G$  vastaa kyselyn  $\phi_G = \exists x(N(s(0)) \wedge N(s(s(x))))$  negaatiota  $\neg\phi_G$ .
- Ajatuksena on, että  $P \cup \{G\}$  on toteutumaton klausuulijoukko  $\iff \Sigma \models \phi_G$ .



### PROLOGin notaatio

- Tyypillisessä PROLOG-toteutuksessa muuttujasymbolit erotetaan vakiosymboleista ison alkukirjamen perusteella.
- Literaalijoukkojen sijaan ohjelma- ja maaliklausuulit kirjoitetaan järjestetyiksi *säännöiksi* seuraavaan tapaan:

$$\{N(0)\} \rightsquigarrow N(0).$$

$$\{N(s(x)), \neg N(x)\} \rightsquigarrow N(s(X)) :- N(X).$$

$$\{\neg N(s(0)), \neg S(s(s(x)))\} \rightsquigarrow :- N(s(0)), N(s(s(X))).$$

- Lisäksi PROLOGissa on käytössä erityinen listanotaatio:
  - $[\ ]$  on tyhjä lista,
  - $[X|Y]$  on lista, joka alkaa alkiolla  $X$  ja jatkuu listana  $Y$ , ja
  - $[X, Y, Z]$  on lista, jossa on alkiot  $X$ ,  $Y$  ja  $Z$ .



## 2.2 Herbrand-malleihin perustuva semantiikka

Tarkastellaan logiikkaohjelmaa (eli ohjelmaklausuulien joukkoa)  $P$ :

- Ohjelman  $P$  Herbrand-universumi  $H$  ja Herbrand-kanta  $B$  määräytyvät siinä esiintyvien symbolien perusteella.
- Olkoon ohjelma  $P'$  ohjelman  $P$  Herbrand-instanssi.
- Herbrand-struktuuri  $\mathcal{H}$  (käytännössä  $B$ :n osajoukko) on ohjelman  $P$  Herbrand-malli  $\iff \mathcal{H}$  on  $P'$ :n Herbrand-malli.

**Esimerkki.** Logiikkaohjelman  $P = \{\{N(0)\}, \{N(s(x)), \neg N(x)\}\}$ , Herbrand-universumi  $U = \{0, s(0), s(s(0)), \dots\}$ , Herbrand-kanta  $B = \{N(0), N(s(0)), N(s(s(0))), \dots\}$  ja Herbrand-instanssi  $P' = \{\{N(0)\}, \{N(s(0)), \neg N(0)\}, \{N(s(s(0))), \neg N(s(0))\}, \dots\}$ .

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



**Väite.** Olkoon  $P$  logiikkaohjelma ja  $B$  vastaava Herbrand-kanta. Tällöin

1. ohjelmalla  $P$  on ns. *pienin Herbrand-malli*  $\mathcal{H}_P \subseteq B$  siten, että  $\mathcal{H}_P \subseteq \mathcal{H}'$  kaikille ohjelman  $P$  Herbrand-malleille  $\mathcal{H}'$ , ja
2.  $\mathcal{H}_P$  on ohjelman  $P$  kaikkien Herbrand-mallien leikkaus.

**Esimerkki.** Logiikkaohjelman  $P = \{\{N(0)\}, \{N(s(x)), \neg N(x)\}\}$  pienin Herbrand-malli  $\mathcal{H}_P = \{N(0), N(s(0)), N(s(s(0))), \dots\}$ .

**Väite.** Olkoon  $P$  logiikkaohjelma ja  $G\theta$  ohjelman  $P$  symboleihin perustuvan maaliklausuulin  $G = \{\neg B_1(\vec{u}_1), \dots, \neg B_m(\vec{u}_m)\}$  muuttujaton instanssi, missä  $\theta$  on ns. *vastaussubstituutio*.

$$\begin{aligned} \text{Nyt } P &\models (B_1(\vec{u}_1) \wedge \dots \wedge B_m(\vec{u}_m))\theta \\ &\iff \mathcal{H}_P \models B_1(\vec{u}_1)\theta \wedge \dots \wedge B_m(\vec{u}_m)\theta \\ &\iff B_1(\vec{u}_1)\theta \in \mathcal{H}_P, \dots, B_m(\vec{u}_m)\theta \in \mathcal{H}_P. \end{aligned}$$

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



## 2.3 Resoluutio logiikkaohjelmien tapauksessa

Jatkossa rajoitetaan resoluutiotodistuksia tehokkuussyistä seuraavasti:

1. Muodostetaan ainoastaan *lineaarisia resoluutiotodistuksia* (eli klausuulien sekvessejä puumaisten todistusten asemesta).
2. Resoluutiotodistukset aloitetaan kyselyn  $\phi_G$  negaatiota  $\neg\phi_G$  vastaavasta maaliklausuulista  $G$  (tavoitehakuisuus).
3. Yksittäisiä klausuuleja käsitellään järjestettyinä literaalien listoina ennemmin kuin literaalien joukkoina.
4. Käytetään valintafunktiota  $R$ , joka valitsee ei-tyhjästä järjestetystä maaliklausuulista  $G$  negatiivisen literaalin  $L = R(G)$ .

Seuraavaksi tarkastellaan näillä rajoituksilla saatavaa todistusmenettelyä (ns. *SLD-resoluutiota*) yksityiskohtaisemmin.



### Resoluutiosääntö järjestetyille klausuuleille

**Määritelmä.** Olkoon  $G = \{\neg B_1(\vec{u}_1), \dots, \neg B_m(\vec{u}_m)\}$  järjestetty maaliklausuuli ja  $C = \{A_1(\vec{t}_1), \neg A_1(\vec{t}_1), \dots, A_n(\vec{t}_n), \neg A_n(\vec{t}_n)\}$  järjestetty ohjelmaklausuuli siten, että

1. klausuuleilla  $G$  ja  $C$  ei ole yhteisiä muuttujia ja
2. atomilla  $A_i(\vec{t}_i)$  ja valintafunktion määräämässä literaalissa  $R(G) = \neg B_i(\vec{u}_i)$  esiintyvällä atomilla  $B_i(\vec{u}_i)$  on yleisin unifioija  $\theta$ .

Klausuulien  $G$  ja  $C$  yhdistelmäksi saadaan järjestetty maaliklausuuli

$$G' = \{ \neg B_1(\vec{u}_1), \dots, \neg B_{i-1}(\vec{u}_{i-1}), \\ \neg A_1(\vec{t}_1), \dots, \neg A_n(\vec{t}_n), \\ \neg B_{i+1}(\vec{u}_{i+1}), \dots, \neg B_m(\vec{u}_m) \} \theta.$$



**Määritelmä.** Olkoon  $P$  joukko järjestettyjä ohjelmaklausuuleita ja  $G$  järjestetty maaliklausuuli. Joukon  $P \cup \{G\}$  hylkäys SLD-resoluutiolla on järjestettyjen maaliklausuulien sekvenssi  $G_1, \dots, G_n$ , missä

1.  $G_1 = G$ ,
2. jokainen järjestetty maaliklausuuli  $G_i$  ( $i > 0$ ) on saatu resoluutiolla edellisestä järjestetystä maaliklausuulista  $G_{i-1}$  ja jostain  $P$ :n järjestetystä ohjelmaklausuulista  $C$  (muuttujat uudelleennimetty) ja
3.  $G_n = \square$ .

**Väite.** (SLD-resoluution virheettömyys ja täydellisyys)

Klausuulijoukolla  $P \cup \{G\}$  on hylkäys SLD-resoluutiolla

$\iff P \cup \{G\}$  on toteutumaton.



## 2.4 PROLOGin hakumekanismi

- PROLOGin perustana on SLD-resoluutio.
- Valintasääntö  $R$  antaa aina järjestetyn maaliklausuulin  $G = \{\neg B_1(\vec{u}_1), \dots, \neg B_m(\vec{u}_m)\}$  ensimmäisen literaalin:  $R(G) = \neg B_1(\vec{u}_1)$ .
- Tyypillisesti PROLOG-tulkki suorittaa palautuvaa syvyyshakua SLD-hylkäyksen  $G_1, \dots, G_n$  löytämiseksi annetusta maaliklausuulista  $G_1 = G$  lähtien.
- Logiikkaohjelman  $P$  klausuuleja käsitellään syötetiedoston määräämässä järjestyksessä  $C_1, \dots, C_n$ .



- Hakuavaruuden pisteet määräytyvät järjestettyjen maaliklausuulien sekvenssien  $G_1, \dots, G_i$  perusteella.
- Toimenpiteet syvyydellä  $i$  olevassa hakuavaruuden pisteessä:
  1. Jos  $G_i = \square$ , haku voidaan todeta onnistuneeksi ja lopettaa.
  2. Jos  $G_i \neq \square$ , tarkastetaan mahdollisuudet tehdä resoluutioaskel maaliklausuulin  $G_i$  ja ohjelmaklausuulien  $C_1, \dots, C_n$  kesken. (Voidaan rajoittaa ohjelmaklausuuleihin  $C_j$ , joissa on literaalia  $R(G_i) = \neg B(\vec{t})$  vastaava positiivinen literaali  $B(\vec{t}')$ ).
  3. Rekursio: mikäli resoluutioaskel on mahdollinen useamman ohjelmaklausuulin  $C'_1, \dots, C'_m$  suhteen, valitaan näistä yksi klausuuli  $C'_j$  kerrallaan, johdetaan resoluutiolla uusi maaliklausuuli  $G_{i+1}$  ja jatketaan hakua syvyydellä  $i + 1$ .



- Koska kysymyksessä on *palautuva* syvyyshaku, hakuavaruuden pisteessä  $G_1, \dots, G_i$  menetellään lisäksi seuraavasti:
  1. Haku voidaan todeta epäonnistuneeksi ko. pisteessä, mikäli
    - $G_i \neq \square$  ja
    - $G_k \neq \square$  kaikille maaliklausuuleille  $G_k$  hakuavaruuden pisteissä  $G_1, \dots, G_i, \dots, G_k$  (missä  $k > i$ ).
  2. Tällöin palataan valintoihin pisteessä  $G_1, \dots, G_{i-1}$  ( $i > 1$ ) **tai** todetaan haku kokonaisuudessaan epäonnistuneeksi ( $i = 1$ ).
- Vastaussubstituutio  $\theta$  saadaan yleisimpien unifioijien kompositiona ja voidaan rajata maaliklausuulissa  $G_1$  esiintyviin muuttujiin.

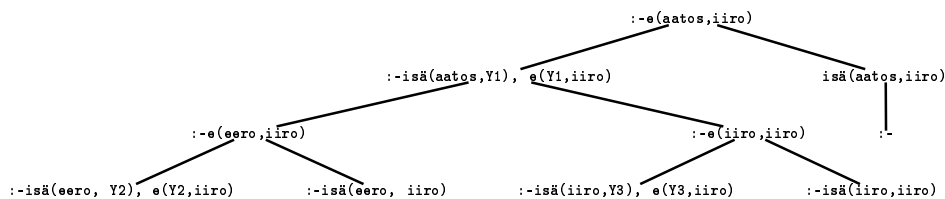


**Esimerkki.** Tarkastellaan seuraavaa logiikkaohjelmaa:

```
isä(aatos,eero). isä(aatos,iiro). isä(oiva,aatos).
```

```
e(X,Z) :- isä(X,Y), e(Y,Z). % esi-isäpredikaatti e
e(X,Y) :- isä(X,Y).
```

Maaliklausuuli :- e(aatos, iiro) johtaa seuraavaan hakuun:



⇒ PROLOG-tulkki vastaa kyselyyn myöntävästi.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio

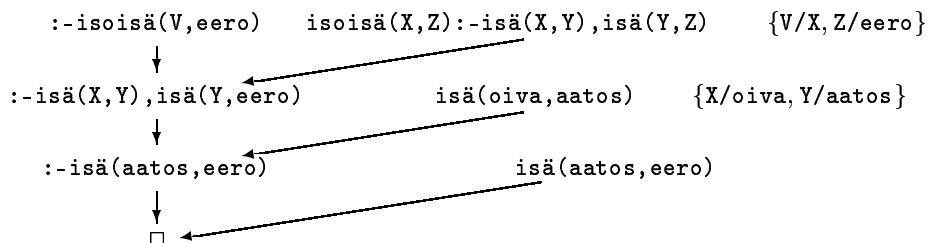


**Esimerkki.** Tarkastellaan edellisen logiikkaohjelman muunnelmaa:

```
isä(aatos,eero). isä(aatos,iiro). isä(oiva,aatos).
```

```
isoisä(X,Z) :- isä(X,Y), isä(Y,Z).
```

Maaliklausuulilla :-isoisä(V,eero) löydetään seuraava SLD-hylkäys.



⇒ Muuttujan V arvo selviää yleisimpien unifioidien kompositiosta:  
{V/oiva, X/oiva, Z/eero, Y/aatos}.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio

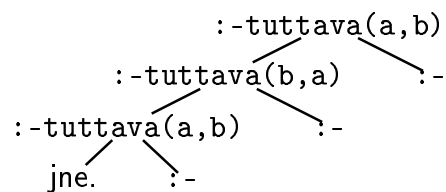


- SLD-resoluution täydellisyys menetetään, Koska PROLOGissa käytetään syvyyshakua (tehokkuussyistä).

**Esimerkki.** Tarkastellaan seuraavia ohjelmaklausuuleja:

```
tuttava(X,Y) :- tuttava(Y,X).  
tuttava(a,b).
```

Maaliklausuulista `:-tuttava(a,b)` aloitettu syvyyshaku ei pääty:



Ongelma vältetään, jos ohjelmaklausuulien järjestys vaihdetaan!



### 3 Ohjelmien oikeellisuustarkastelut

- Ehtolausekkeiden ekvivalenssi
- Toistolausekkeiden invariantit
- Rakenteinen induktio ja rekursiiviset tietotyypit





### 3.1 Ehtolausekkeiden ekvivalenssi

- Ohjelmointikielissä käytetään paljon ehtolausekkeitä kontrolloimaan, millä ehdoilla ja mitä toimintoja suoritetaan.
- Jos ehtolausekkeitä muutetaan esim. optimointitarkoituksessa, halutaan varmistua että toiminnot suoritetaan samoilla ehdoilla.
- Ehtolausekkeiden ekvivalenssin osoittamiseen voidaan käyttää sekä lauselogiikkaa tai predikaattilogiikkaa.
- Jos ehtolausekkeilla on *sivuvaikutuksia*, pelkkä loogisen ekvivalenssin tarkastaminen ei välttämättä riitä.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



**Esimerkki.** Tarkastellaan johdantoluennon esimerkkiä:

```
P1: if(myptr != NULL && myptr[0] == '/') doit(myptr);  
    if(!(myptr == NULL || myptr[0] == '.')) dothat(myptr);
```

```
P2: if(myptr != NULL) {  
    if(myptr[0] == '/') doit(myptr);  
    if(myptr[0] != '.') dothat(myptr);  
}
```

- Valitaan seuraavat atomiset lauseet:  $A = \text{"myptr == NULL"}$ ,  $B = \text{"myptr[0] == '/'}$  ja  $C = \text{"myptr[0] == '.'}$ .
- Nyt esim. proseduuria `dothat(myptr)` kutsutaan ehdoilla  $\neg(A \vee C)$  (P1) ja  $\neg A \wedge \neg C$  (P2), jotka ovat ekvivalentit lauselogiikan nojalla.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



### 3.2 Toistolausekkeiden invariantit

- Ohjelmointikielten keskeisiä primitiivejä ovat erilaiset toistolausekkeet, joiden avulla jotain toimintoa (proseduuria) voidaan toistaa haluttu määrä:  

```
for(i=0; i<n; i++) printf("%i ", i);  
while(i<n) if((k=a[i++])>m) m=k;
```
- Ongelma: kuinka voitaisiin osoittaa toistorakenteita sisältävien algoritmien toimivuus kaikissa tilanteissa?
- Toistorakenteelle halutaan tyypillisesti osoittaa *invariantti*  $\phi$  eli ominaisuus, joka säilyy voimassa toistorakenteen suorituksen ajan.
- Tällöin on luontevaa käyttää induktiota toistorakenteen suorituskertojen lukumäärän suhteen.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



**Esimerkki.** Tarkastellaan seuraavaa C-kielistä funktiota ja siinä esiintyvää while-silmukkaa.

```
int sum(int a[], int n)    /* taulukko ja taulukon koko */  
{  
    int s=0;              /* taulukon lukujen summa */  
    int i=0;              /* indeksi */  
    while(i<n) {  
        s = s+a[i]; i = i+1; /* summaus */  
    }  
    return s;  
}
```

Jatkossa osoitetaan, että funktio palauttaa mielivaltaisen kokonaislukutaulukon alkioden summan.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



- Todistetaan induktiolla silmukan suorituskertojen ( $k$ ) suhteen, että:

$$s_k = \sum_{j=0}^{j < i_k} a[j] \text{ ja } i_k \leq n,$$

missä  $s_k$  ja  $i_k$  ovat muuttujien  $s$  ja  $i$  arvot, kun funktion `sum` while-silmukkaa on suoritettu  $k$  kertaa.

- *Perustapaus:* ( $k = 0$  eli silmukkaa ei ole suoritettu kertaakaan):

$$s_0 = 0 = \sum_{j=0}^{j < i_0} a[j] = \sum_{j=0}^{j < 0} a[j] \text{ ja } i_0 = 0 \leq n.$$

- *Induktiohypoteesi* (tilanne  $k$ :n suorituskerran jälkeen):

$$s_k = \sum_{j=0}^{j < i_k} a[j] \text{ ja } i_k \leq n.$$

- *Induktioaskel* (huomaa, että  $i_k < n$  silmukan ehdon perusteella):

$$s_{k+1} = s_k + a[i_k] = \sum_{j=0}^{j < i_k} a[j] + a[i_k] = \sum_{j=0}^{j < i_{k+1}} a[j] \text{ ja} \\ i_{k+1} = i_k + 1 \leq n.$$



- Tyypillisesti pelkkä invariantti ei riitä takaamaan, että toistorakenteen suorittaminen johtaisi lopulta haluttuun tulokseen.
- Toistorakenteen lopetusehdosta saadaan usein kriittistä lisäinformaatiota.

**Esimerkki.** Jatketaan edellisen esimerkin `sum`-funktion analysointia.

1. Silmukan suoritus päättyy, kun  $i_k < n$  tulee epätodeksi.
2. Koska tällöin  $i_k \leq n$  on edelleen tosi, tiedämme, että  $i_k = n$ .
3. Tällöin muuttujan  $s$  arvona on

$$s_k = \sum_{j=0}^{j < n} a[j],$$

mikä onkin taulukon  $a$  lukujen summa.



### 3.3 Rakenteinen induktio ja rekursiiviset tietotyypit

- Tietojenkäsittelyssä käytetään paljon rekursiivisesti määriteltyjä tietorakenteita: listat, pinot, puut, jne.
- Induktiota voidaan käyttää myös tällaisia tietorakenteita käsittelevien ohjelmien oikeellisuustarkasteluissa.
- Tietyissä tapauksissa induktio voidaan kytkeä suoraan tietorakenteen induktiiviseen määritelmään: esim. puiden lehti- ja sisäsolmujen käsittely (*rakenteinen induktio*).
- Induktio voidaan suorittaa myös jonkin monotonisesti kasvavan *mitan* (esim. listan pituus tai puun korkeus) suhteen.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



**Esimerkki.** Tarkastellaan PROLOG-proseduuria `reverse`, joka järjestää listan käänteiseen järjestykseen (listasta `[1, 2, 3]` saadaan `[3, 2, 1]`):

```
reverse([], []).
```

```
reverse([X|Y], Z) :- reverse(Y, V), append(V, X, Z).
```

```
append([], X, [X]).
```

```
append([X|Y], Z, [X|V]) :- append(Y, Z, V).
```

#### Huomioita.

- Toteutuksessa on käytetty apuproseduuria/apupredikaattia `append` (jonka oikeellisuuden joudumme osoittamaan samalla).
- Oikeellisuustarkastelussa osoitetaan induktiolla, että `reverse` käsittelee pituudeltaan  $n$  olevat listat oikein.

© 2001 Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio



- PROLOG-tulkki vastaa myönteisesti ja palauttaa vastaussubstituution  $\theta$ , mikäli tulkki onnistuu johtamaan vastaavasta maaliklausuulista tyhjän klausuulin  $\square$  ( $\theta$  on yleisimpien unifioidien kompositio).

**Todistettavat ominaisuudet:**

1. PROLOG-tulkki antaa kyselyyn `-reverse(L,M)` vastaussubstituution  $\theta$   
 $\iff$  lista  $M\theta$  on lista  $L$  käännettynä.
  2. PROLOG-tulkki antaa kyselyyn `-append(L,A,M)` vastaussubstituution  $\theta$   
 $\iff$  listassa  $M\theta$  esiintyvät ensin listan  $L$  alkio ja sitten alkio  $A$ .
- Yllä  $L$  ja  $M\theta$  sekä alkio  $A$  ovat muuttujattomia ja listan  $L$  pituus on  $n$ .



- Osoitetaan induktiolla, että kaikilla luonnollisilla luvuilla  $n$  on em. ominaisuus (joten PROLOG-proseduurit `reverse` ja `append` käsittelevät oikein mielivaltaisen pitkiä listoja  $L$ ).
- *Perustapaus* ( $n = 0$ ):  
Koska  $n = 0$ , listan  $L$  täytyy olla tyhjä lista  $[\ ]$ .  
Tällöin kysely `-reverse([\ ],M)` palauttaa vastauksen  $\theta$   
 $\iff M\theta = [\ ]$  ( $\theta$  on termien  $M$  ja  $[\ ]$  MGU)  
 $\iff$  lista  $M\theta$  on lista  $[\ ]$  käännettynä.  
Vastaavasti kysely `-append([\ ],A,M)` palauttaa vastauksen  $\theta$   
 $\iff M\theta = [A]$  ( $\theta$  on termien  $M$  ja  $[A]$  MGU)  
 $\iff$  listassa  $M\theta$  esiintyy ensin listan  $[\ ]$  alkio ja sitten alkio  $A$ .
- *Induktiohypoteesi*:  
todistettava ominaisuus on voimassa, kun listan  $L$  pituus on  $n$ .



*Induktioaskel:* Olkoon listan  $L$  pituus  $n + 1$ .

Kysely :  $-reverse(L, M)$  palauttaa vastauksen  $\theta = \theta_1\theta_2\theta_3$

$\iff \theta_1$  on atomien  $reverse(L, M)$  ja  $reverse([X|Y], Z)$  MGU,  
 $L = [X|Y]\theta_1 = [X\theta_1|Y\theta_1]$  ja  $M\theta_1 = Z\theta_1$ ,

kysely :  $-reverse(Y\theta_1, V)$  palauttaa vastauksen  $\theta_2$  ja

kysely :  $-append(V\theta_2, X\theta_1, Z\theta_1)$  palauttaa vastauksen  $\theta_3$

$\iff \theta_1$  on atomien  $reverse(L, M)$  ja  $reverse([X|Y], Z)$  MGU,  
 $L = [X|Y]\theta_1 = [X\theta_1|Y\theta_1]$  ja  $M\theta_1 = Z\theta_1$ ,

lista  $V\theta_2$  on lista  $Y\theta_1$  käännettynä ja listassa

$Z\theta_1\theta_3$  esiintyy ensin listan  $V\theta_2$  alkio ja sitten alkio  $X\theta_1$

$\iff$  lista  $M\theta = M\theta_1\theta_3 = Z\theta_1\theta_3$  on lista  $L = [X\theta_1|Y\theta_1]$  käännettynä.

**Huomio.** Yllä  $L$ ,  $X\theta_1$ ,  $Y\theta_1$ ,  $V\theta_2$  ja  $Z\theta_1\theta_3$  ovat muuttujattomia.



Kysely :  $-append(L, A, M)$  palauttaa vastauksen  $\theta = \theta_1\theta_2$

$\iff \theta_1$  on atomien  $append(L, A, M)$   
ja  $append([X|Y], Z, [X|V])$  MGU,  
 $L = [X|Y]\theta_1 = [X\theta_1|Y\theta_1]$ ,  $A = Z\theta_1$  ja  $M\theta_1 = [X|V]\theta_1 = [X\theta_1|V\theta_1]$

ja kysely :  $-append(Y\theta_1, Z\theta_1, V\theta_1)$  palauttaa vastauksen  $\theta_2$

$\iff \theta_1$  on atomien  $append(L, A, M)$   
ja  $append([X|Y], Z, [X|V])$  MGU,  
 $L = [X|Y]\theta_1 = [X\theta_1|Y\theta_1]$ ,  $A = Z\theta_1$  ja  $M\theta_1 = [X|V]\theta_1 = [X\theta_1|V\theta_1]$

listassa  $V\theta_1\theta_2$  esiintyy listan  $Y\theta_1$  alkioiden jälkeen alkio  $Z\theta_1$ .

$\iff$  listassa  $M\theta = M\theta_1\theta_2 = [X\theta_1|V\theta_1\theta_2]$  esiintyy ensin listan  
 $L = [X\theta_1|Y\theta_1]$  alkio ja sitten alkio  $Z\theta_1 = A$ .

**Huomio.** Yllä  $L$ ,  $A$ ,  $X\theta_1$ ,  $Y\theta_1$ ,  $Z\theta_1$ ,  $V\theta_1\theta_2$  ovat muuttujattomia.



## Haluatko tietää logiikasta lisää ???

Muita laboratoriomme järjestämiä logiikkaan liittyviä kursseja:

- Tik-79.146 Logiikka tietotekniikassa: erityiskysymyksiä I (kl)
- Tik-79.154 Logiikka tietotekniikassa: erityiskysymyksiä II (sl)
- Tik-79.240 Laskennallisen vaativuuden erikoiskurssi (sl)

Näillä opintojaksoilla käsitellään lisää logiikkaan liittyviä aiheita.