Logiikan sovelluskohteita

Voidaan nähdä karkea kahtiajako:

- **Päättelykomponentti järjestelmän osana**
  Toteutetaan loogista päättelyä jollain tasolla
  Logiikkaohjelmointi, rajoiteohjelmointi, sääntöpohjainen päättely
  Automaattinen teoreemantodistaminen
- **Järjestelmän ominaisuuksien analysointi (metataso)**
  Ohjelmien oikeellisuustarkastelut
  Tarkistetaan, että järjestelmä toteuttaa annetun spesifikaation.
  Osoitetaan, että spesifikaation mukaisella järjestelmällä on halutut ominaisuudet.

Paketti III: ESIMERKKEJÄ SOVELLUKSISTA

1. Teoreemantodistin OTTER
2. Logiikkaohjelmointi
3. Sääntöpohjainen päättely
4. Induktioperiaatteen käyttökohteita

1 Teoreemantodistin OTTER

Syntaksi:	$(\forall x\ \varphi)$	`(all x (`φ`))`
	$(\exists x\ \varphi)$	`(exists x (`φ`))`
	$(\varphi \wedge \psi)$	`(`φ `&` ψ`)`
	$(\varphi \vee \psi)$	`(`φ `|` ψ`)`
	$(\varphi \rightarrow \psi)$	`(`φ `->` ψ`)`
	$(\varphi \leftrightarrow \psi)$	`(`φ `<->` ψ`)`
	$(t_1 = t_2)$	`(`t_1 `=` t_2`)`
	$(\neg\varphi)$	`(-`φ`)`

Esim.

$\forall x \exists y\ P(x,y)$	`(all x (exists y P(x,y))).`
$\forall x(P(x) \wedge (Q(x) \vee R(x)))$	`(all x (P(x) & (Q(x)	R(x)))).`

Suositeltavat asetukset

```
set(auto).

formula_list(usable).

% tähän tulee sitten tarvittava lausejoukko

% ja kyselyn negaatio (vain yksi kysely kerrallaan)

end_of_list.
```

Esimerkki. (Avioliittoesimerkki OTTERilla)

```
set(auto).
formula_list(usable).

(all x (nainen(x) | mies(x))).
-(exists x (mies(x) & nainen(x))).

(all x -(aviossa(x,x))).
(all x y (aviossa(x,y) -> aviossa(y,x))).
(all x y z (aviossa(x,y) & aviossa(x,z) -> y=z)).
(all x y (aviossa(x,y) ->
          (nainen(x) & mies(y)) | (mies(x) & nainen(y)) )).

-(ann=eve). -(aki=eve). -(aki=ann).
(all x (x=aki | x=ann | x=eve)).
nainen(ann). nainen(eve). mies(aki). aviossa(aki,ann).

-(-aviossa(eve,aki)).     % !!! Kyselyn _negaatio_ !!!
end_of_list.
```


OTTERin tulosteet

```
usable clausifies to:   list(usable).
                        0 [] nainen(x)|mies(x).
                        0 [] -mies(x)| -nainen(x).
                        0 [] -aviossa(x,x).
                        0 [] -aviossa(x,y)|aviossa(y,x).
                        0 [] -aviossa(x,y)| -aviossa(x,z)|y=z.
                        0 [] -aviossa(x,y)|nainen(x)|mies(x).
                        0 [] -aviossa(x,y)|nainen(x)|nainen(y).
                        0 [] -aviossa(x,y)|mies(y)|mies(x).
                        0 [] -aviossa(x,y)|mies(y)|nainen(y).
                        0 [] ann!=eve.
                        0 [] aki!=eve.
                        0 [] aki!=ann.
                        0 [] x=aki|x=ann|x=eve.
                        0 [] nainen(ann).
                        0 [] nainen(eve).
                        0 [] mies(aki).
                        0 [] aviossa(aki,ann).
                        0 [] aviossa(eve,aki).
                        end_of_list.
```

```
------------> process usable:
** KEPT (pick-wt=4): 1 [] -mies(x)| -nainen(x).
** KEPT (pick-wt=3): 2 [] -aviossa(x,x).
** KEPT (pick-wt=6): 3 [] -aviossa(x,y)|aviossa(y,x).
** KEPT (pick-wt=9): 4 [] -aviossa(x,y)| -aviossa(x,z)|y=z.
** KEPT (pick-wt=7): 5 [] -aviossa(x,y)|nainen(x)|mies(x).
** KEPT (pick-wt=7): 6 [] -aviossa(x,y)|nainen(x)|nainen(y).
** KEPT (pick-wt=7): 7 [] -aviossa(x,y)|mies(y)|mies(x).
** KEPT (pick-wt=7): 8 [] -aviossa(x,y)|mies(y)|nainen(y).
** KEPT (pick-wt=3): 10 [copy,9,flip.1] eve!=ann.
** KEPT (pick-wt=3): 12 [copy,11,flip.1] eve!=aki.
** KEPT (pick-wt=3): 14 [copy,13,flip.1] ann!=aki.
------------> process sos:
** KEPT (pick-wt=4): 16 [] nainen(x)|mies(x).
** KEPT (pick-wt=9): 17 [] x=aki|x=ann|x=eve.
** KEPT (pick-wt=2): 18 [] nainen(ann).
** KEPT (pick-wt=2): 19 [] nainen(eve).
** KEPT (pick-wt=2): 20 [] mies(aki).
** KEPT (pick-wt=3): 21 [] aviossa(aki,ann).
** KEPT (pick-wt=3): 22 [] aviossa(eve,aki).
16 back subsumes 8.
16 back subsumes 5.
```



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

given clause #1: (wt=4) 16 [] nainen(x)|mies(x).
given clause #2: (wt=2) 18 [] nainen(ann).
given clause #3: (wt=2) 19 [] nainen(eve).
given clause #4: (wt=2) 20 [] mies(aki).
given clause #5: (wt=3) 21 [] aviossa(aki,ann).
given clause #6: (wt=9) 17 [] x=aki|x=ann|x=eve.
given clause #7: (wt=3) 22 [] aviossa(eve,aki).
given clause #8: (wt=3) 26 [hyper,21,15] ann=ann.
given clause #9: (wt=3) 27 [hyper,21,3] aviossa(ann,aki).
given clause #10: (wt=3) 30 [hyper,17,12,unit_del,10] eve=eve.
given clause #11: (wt=8) 23 [para_into,18.1.1,4.3.1]
                            nainen(x)|-aviossa(y,ann)|-aviossa(y,x).
given clause #12: (wt=3) 31 [para_from,17.2.1,21.1.1,unit_del,2,12]
                            aki=aki.
given clause #13: (wt=3) 32 [hyper,22,3] aviossa(aki,eve).

----> UNIT CONFLICT at   0.06 sec ----> 40 [binary,38.1,10.1] $F.
```

```
---------------- PROOF ----------------

3 [] -aviossa(x,y)|aviossa(y,x).
4 [] -aviossa(x,y)| -aviossa(x,z)|y=z.
9 [] ann!=eve.
10 [copy,9,flip.1] eve!=ann.
21 [] aviossa(aki,ann).
22 [] aviossa(eve,aki).
32 [hyper,22,3] aviossa(aki,eve).
38 [hyper,32,4,21,flip.1] eve=ann.
40 [binary,38.1,10.1] $F.

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

Search stopped by max_proofs option.
```


- Huomaa, että jos OTTER ei löydä todistusta, OTTER voi pysähtyä (ja ilmoittaa "search stopped because sos is empty") tai jäädä ikuiseen silmukkaan. Tämä käyttäytyminen liittyy predikaattilogiikan puoliratkeavuuteen. Esimerkissämme ilmiö tulee esille muuttamalla kyselyksi `-aviossa(aki,ann)`.
- Muuttujasidontojen eristämistä varten OTTERissa voi määritellä `$ans`-alkuisia predikatteja, joiden argumenteiksi tulee mielenkiinnon kohteena olevat muuttujat. Esim. kyselyllä `(exists x (-aviossa(aki,x) & $ans(x)))` saadaan todistus

  ```
  2 [] -aviossa(x,x).
  22 [] aviossa(aki,x)| -$ans(x).
  23 [binary,22.1,2.1] -$ans(aki).
  ```

  Tästä tiedämme, että aki on vaatimuksen täyttävä henkilö.

OTTER TKK:n koneissa

- Asennettuna seuraavissa koneissa: alpha, beta, gamma, ...
- Otetaan käyttöön kirjoittamalla: `use otter`
  (`source /p/setup/use/shell/use.csh`)
- Käynnistäminen: kirjoitetaan lausejoukko tiedostoon xyz.in
  käynnistetään OTTER kirjoittamalla

  `otter < xyz.in`
- Hakemistossa /p/edu/tik-79.144: esimerkkejä (matematiikasta pääasiassa), MS-DOS -binääri, Macintosh-binääri, yms.
- Manuaaliin löytyy linkki kurssin WWW-sivulta

2 Logiikkaohjelmointi

- *Horn-klausuuli* on klausuuli, jossa on enintään yksi positiivinen literaali.

  Esim. $\{A\}$, $\{A, \neg B, \neg C\}$, $\{\neg D, \neg E, \neg F\}$.

  Prologin notaatiolla: `A.` `A :- B, C.` `:- D, E, F.`
- *Ohjelmaklausuuli* on klausuuli, jossa on täsmälleen yksi positiviinen literaali.

  Esim. $\{A\}$, $\{A, \neg B, \neg C\}$
- *Maaliklausuuli* on klausuuli jossa on vain negatiivisia literaaleja.

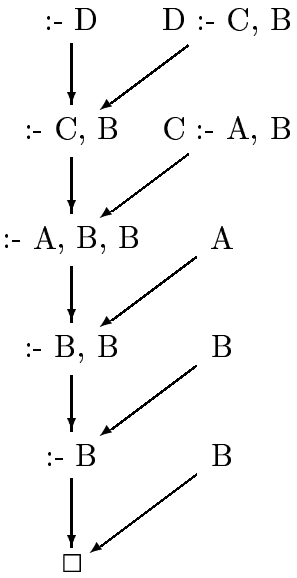
  Esim. $\{\neg D, \neg E, \neg F\}$
- *Prolog-ohjelma* on joukko ohjelmaklausuuleja.

LI/LD/SLD resoluutio

L=Linear
I=Input
D=Definite
S=Selection Function

```
A.
B.
C :- A, B.
D :- C, B.
```

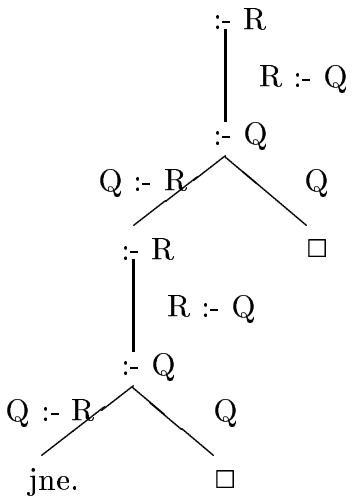
Kysely `:- D`



Väite. Jos S on toteutumaton Prolog-ohjelma (joukko ohjelmaklausuuleja + maaliklausuuli), niin S:lle on olemassa SLD-refutaatio.

- Tämä täydellisyysominaisuus menetetään Prologissa, jos käytetään syvyyshakua (kts. seuraava esimerkki).

Prologin hakuproseduurin epätäydellisyys



R :- Q.

Q :- R.

Q.

Kyselynä :- R.

Prolog-ohjelmat predikaattilogiikassa

Prolog	Klausuuliesitys	Vastaava lause
`p(X)`	$\{P(x)\}$	$\forall x(P(x))$
`r(X,Y) :- q(X)`	$\{R(x,y), \neg Q(x)\}$	$\forall x \forall y(Q(x) \rightarrow R(x,y))$
`:-q(X),s(X)`	$\{\neg Q(x), \neg S(x)\}$	$\neg \exists x(Q(x) \wedge S(x))$

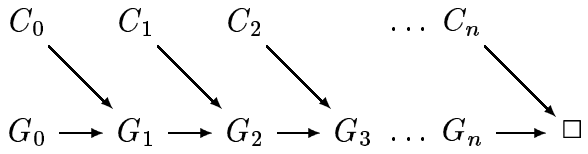
- Ohjelmaklausuulien muuttujat ovat universaalisti kvantifioidut ja kyselyn muuttujat ovat eksistentiaalisesti kvantifioidut.

Listanotaatio:

Prolog	Vastaava termi
`[]`	e
`[X|[Y|Z]]`	$\mathrm{c}(x, \mathrm{c}(y, z))$
`[X,Y,Z]`	$\mathrm{c}(x, \mathrm{c}(y, \mathrm{c}(z, e)))$

Lineaarinen input-resoluutio

Olkoon P joukko klausuuleja ja G kysely (sisältää vain negatiivisia literaaleja). Allaoleva on refutaatio *lineaarisella input-resoluutiolla* (LI), joss $G_0 = G$ ja kukin C_i on P:n klausuuli, missä muuttujat on tarvittaessa uudelleennimetty.



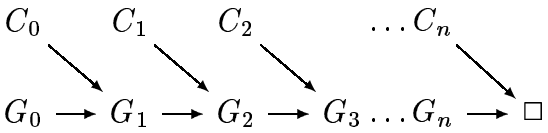
Väite. (Virheettömyys ja täydellisyys) Olkoon P joukko ohjelmaklausuuleja ja G kysely. Nyt $P \cup \{G\}$ on toteutumaton, jos ja vain jos on olemassa $P \cup \{G\}$:n refutaatio lineaarisella input-resoluutiolla siten, että refutaatio alkaa klausuulista $G_0 = G$.

Todistus: (Virheettömyys) Resoluutioaskeleet säilyttävät toteutuvuuden. (Täydellisyys) Lineaarinen resoluutio takaa refutaation olemassaolon (T II.14.4). Klausuuleissa G_i on vain negatiivisia literaaleja, joten jokaisella askeleella klausuuli C_i on välttämättä ohjelmaklausuuli (jossa on siis täsmälleen yksi positiivinen literaali).

Järjestetyt Horn-klausuulit

Olkoot $G = \{\neg A_0, \ldots, \neg A_n\}$ ja $C = \{B, \neg B_0, \ldots, \neg B_m\}$ järjestettyjä klausuuleita, ja θ literaalien A_i ja B yleisin unifioija. Klausuuleista G ja C saadaan resoluutiolla järjestetty klausuuli
$G = \{\neg A_0, \ldots, \neg A_{i-1}, \neg B_0, \ldots, \neg B_m, \neg A_{i+1}, \ldots, \neg A_n\}\theta$.

Olkoon P joukko järjestettyjä klausuuleita ja G järjestetty kysely. Allaoleva on refutaatio *LD-resoluutiolla* (Linear Definite), jos $G_0 = G$ ja kukin C_i on P:n klausuuli, missä muuttujat on tarvittaessa uudelleennimetty.



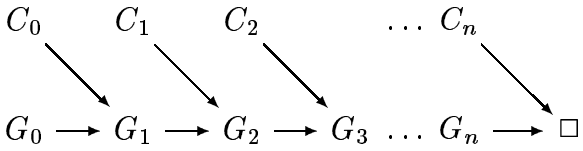
LD-resoluutio valintasäännöllä = SLD-resoluutio

Valintasääntö on funktio R, joka valitsee järjestetystä ei-tyhjästä klausuulista C literaalin $L = R(C)$.

Prologin valintasääntö R_P:

Klausuulille $C = \{\neg A_1, \ldots, \neg A_n\}$, $R_P(C) = \neg A_1$.

Olkoon P joukko järjestettyjä klausuuleita ja G järjestetty kysely. Allaoleva on refutaatio *SLD-resoluutiolla* (Linear Definite), jos $G_0 = G$ ja kukin C_i on P:n klausuuli (muuttujat uudelleennimetty), ja *askeleella i resoluutiosääntöä sovellettiin klausuuleihin G_i ja C_i literaaliin $R(G_i)$.*

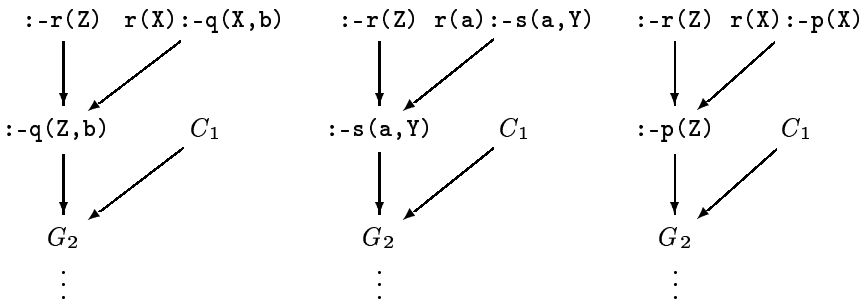


Tämä on edelleen virheetön ja täydellinen resoluutiomenetelmä (suhteessa $P \cup \{G\}$ toteutumattomuuteen).

Prologin hakumekanismi

```
r(X) :- q(X,b).
r(a) :- s(a,Y).
r(X) :- p(X).
...
```

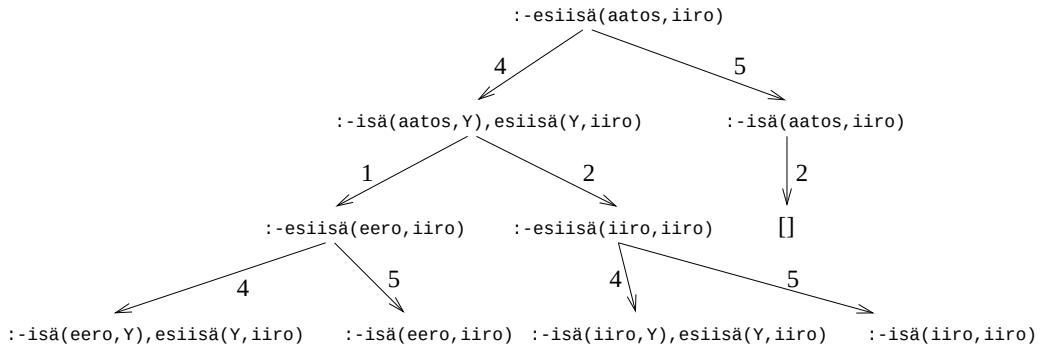
Kysely: `:-r(Z)`



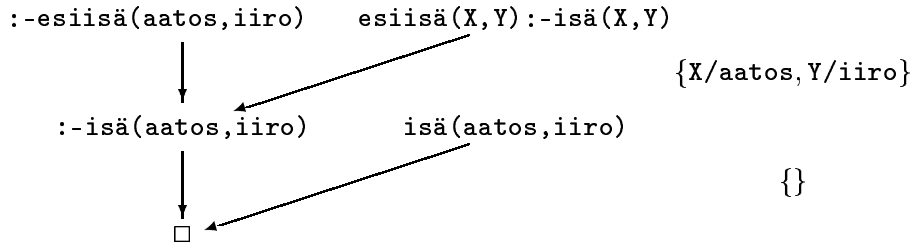
Esimerkki

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

esiisä(X,Z) :- isä(X,Y), esiisä(Y,Z).
esiisä(X,Y) :- isä(X,Y).

isoisä(X,Z) :- isä(X,Y), isä(Y,Z).


:- esiisä(aatos, iiro) ?
```

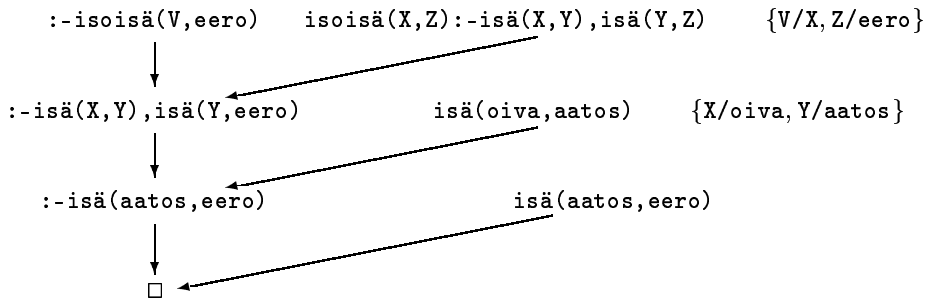

Hakupuu kyselylle



Tuloksena saatava lineaarinen resoluutiotodistus

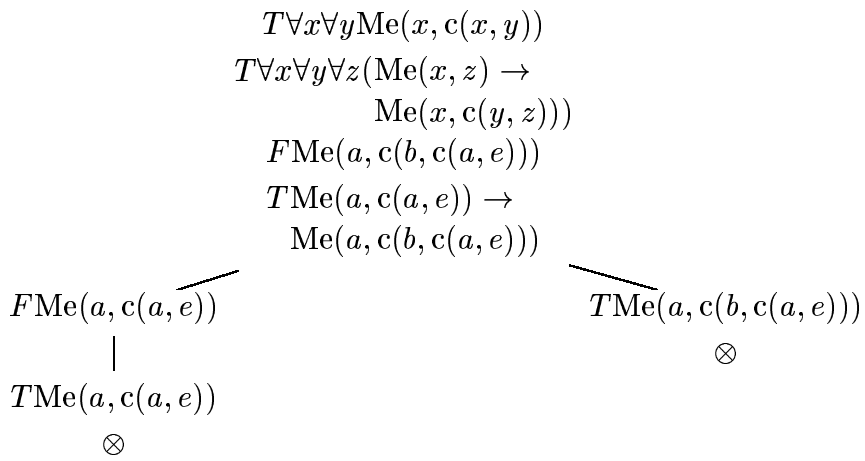


Vastauksena sidokset kyselyn muuttujille



Unifioijien kompositio: {V/oiva, X/oiva, Z/eero, Y/aatos}

Esimerkki

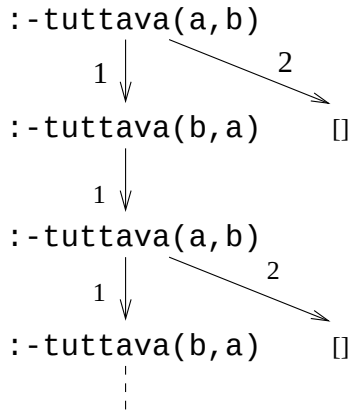



```
Me(X,[X|Y]).    Me(X,[Y|Z]) :- Me(X,Z).    :- Me(a,[b,a])
```


Syvyyshaku: epätäydellisyys

```
1: tuttava(X,Y) :- tuttava(Y,X)
```
Kysely: `:-tuttava(a,b)`

```
2: tuttava(a,b)
```



Prologin not-operaattori

- Halutaan käyttää muunkinlaisia sääntöjä kuin $P(t_1, \ldots, t_n)$ ja $P(t_1, \ldots, t_n)$:- $P_1(s_{11}, \ldots, s_{1n_1}), \ldots, P_m(s_{m1}, \ldots, s_{mn_m})$. Esim. ei-operaatiota säännön ehto-osassa.
- Koko predikaattilogiikka liian monimutkainen: ohjelmien suorittaminen ei olisi kyllin tehokasta
- Laajennetaan resoluutioproseduuria:
  - Käytetään predikaattilogiikan negaation "$\neg$" asemesta toisentyyppistä negaatiota "`not`".
  - Tulokset käytännön sovelluksissa useimmiten haluttuja

NOT – negaatio epäonnistumisena

"Jos ei voida päätellä ϕ, päätellään `not` ϕ"

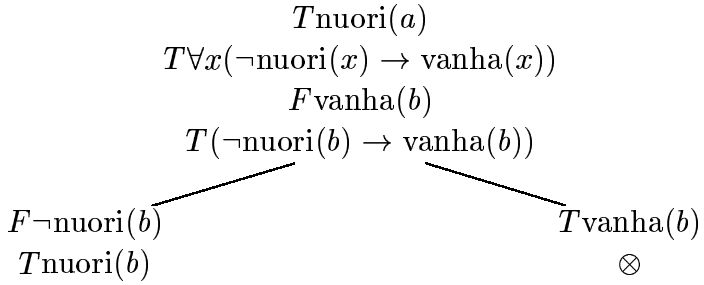
```
koululainen(oskari).
koululainen(emilia).
sairas(oskari).
koulussa(X) :- koululainen(X), not sairas(X).
```

1. Kysely: `:-koulussa(oskari)`
2. Kysely: `:-koulussa(emilia)`

Ero klassiseen negaatioon

```
nuori(a).
vanha(X) :- not nuori(X).
```

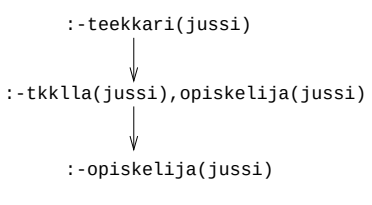
Prolog vastaa kyselyyn vanha(b) myönteisesti. Lause vanha(b) ei ole kuitenkaan ohjelman looginen seuraus, jos klausuuleissa esiintyvä Prologin negaatio tulkitaan predikaattilogiikan negaatioksi:



Määritelmä. *Äärellinen epäonnistunut SLD-puu* P:lle G:lle ja R:lle on äärellinen SLD-resoluution hakupuu, jonka mikään lehti ei ole tyhjä klausuuli $\square$.

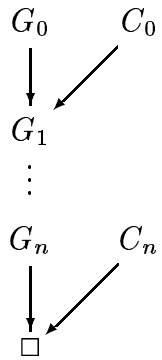
Äärellinen epäonnistunut SLD-puu kyselylle `teekkari(jussi)`:

```
teekkari(X) :- tkklla(X),
               opiskelija(X).
opiskelija(teemu).
tkklla(jussi).
tkklla(teemu).
```



SLDNF-resoluutio

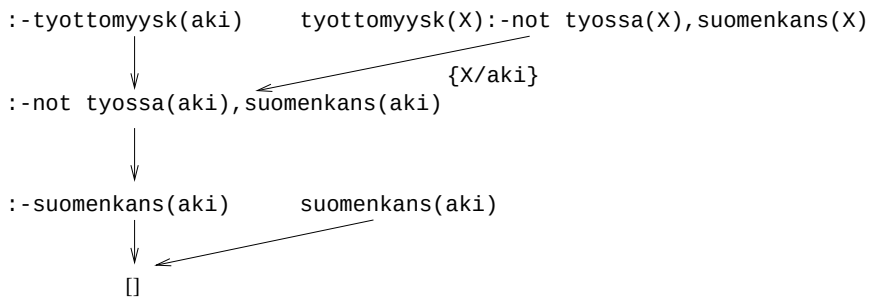
Olkoon P joukko järjestettyjä klausuuleita ja G järjestetty kysely. Vieressä oleva on refutaatio *SLDNF-resoluutiolla* (Selection function, Linear Definite, Negation as Failure), jos $G_0 = G$ ja kukin C_i on P:n klausuuli (mahdollisesti muuttujat uudelleennimettynä), ja



- $R(G_i)$ on not L, L on muuttujaton, ja L:llä P:llä ja R:llä on äärellinen epäonnistunut SLDNF-puu ja G_{i+1} on $G_i - \{\text{not } L\}$, tai
- askeleella i suoritettiin resoluutio klausuuleille G_i ja C_i literaalilla $R(G_i)$.


```
suomenkans(aki).
suomenkans(eve).
tyossa(eve).
tyottomyysk(X) :- not tyossa(X), suomenkans(X).
```

Kyselylle `:-tyottomyysk(aki)` saadaan seuraava SLDNF-refutaatio:



Huomaa, että alikysely `:-tyossa(aki)` epäonnistuu.

Prolog-ohjelman täydentäminen

$$P(t_{11}, \ldots, t_{1n}) \text{ :- } l_{11}, \ldots, l_{1m_1}$$
$$\vdots$$
$$P(t_{r1}, \ldots, t_{rn}) \text{ :- } l_{r1}, \ldots, l_{rm_r}$$

P-predikaatin klausuulit täydenetään lauseeksi

$$\begin{aligned}\forall x_1 \cdots \forall x_n \; (P(x_1, \ldots, x_n) \leftrightarrow \\ (\exists y_{11} \cdots \exists y_{1v_1}(x_1 = t_{11} \wedge \cdots \wedge x_n = t_{1n} \wedge l_{11} \wedge \cdots \wedge l_{1m_1}) \vee \\ \cdots \\ \vee \exists y_{r1} \cdots \exists y_{rv_r}(x_1 = t_{r1} \wedge \cdots \wedge x_n = t_{rn} \wedge l_{r1} \wedge \cdots \wedge l_{rm_r})))\end{aligned}$$

Ohjelmaa P vastaava lausejoukko Comp(P) sisältää täydennettyjen klausuulien lisäksi yksikäsitteisten nimien ja sulkeuma-aksiomat.

Esimerkki. Ohjelma P:

```
suomenkansalainen(aki).
suomenkansalainen(eve).
tyossa(eve).
tyottomyyskorvausta(X) :- not tyossa(X),
                          suomenkansalainen(X).
```

on täydennettynä

$$\begin{aligned}&\forall x \; (\text{suomenkansalainen}(x) \leftrightarrow (x = \text{aki} \vee x = \text{eve})) \\ &\forall x \; (\text{tyossa}(x) \leftrightarrow (x = \text{eve})) \\ &\forall x \; (\text{tyottomyyskorvausta}(x) \leftrightarrow \\ &\qquad (\neg \text{ tyossa}(x) \wedge \text{ suomenkansalainen}(x))) \\ &\neg(\text{aki} = \text{eve})\end{aligned}$$

Nyt Comp(P) $\models$ tyottomyyskorvausta(aki).

Väite. Jos not-operaatiota sisältävällä ohjelmalla P ja kyselyllä G = $\{\neg L_1, \ldots, \neg L_m\}$ on SLDNF-refutaatio, niin

$$\text{Comp(P)} \models \exists x_1 \cdots \exists x_n (L_1 \wedge \cdots \wedge L_m).$$

Esimerkki. Kaikkia Comp(P):n loogisia seurauksia ei saada todistetuksi SLDNF-resoluutiolla:

```
v(X) :- not q(X).
q(a) :- not r(a).
r(a) :- p(a).
r(a) :- not p(a).
p(X) :- p(f(X)).
```

$\forall x(v(x) \leftrightarrow \neg q(x))$

$q(a) \leftrightarrow \neg r(a)$

$r(a) \leftrightarrow (p(a) \vee \neg p(a))$ eli $r(a)$

$\forall x(p(x) \leftrightarrow p(f(x)))$

Kyselyyn `:-v(a)` ei saada vastausta, vaikka Comp(P) $\models v(a)$.

Jos negaation yhteydessä käytetään muuttujia sisältäviä literaaleja, SDLNF-resoluution virheettömyys menetetään (floundering):

```
koululainen(oskari).
sairas(liisa).
koulussa(X) :- not sairas(X), koululainen(X).
```

Kysely `:-koulussa(X)` epäonnistuu.

$$\begin{aligned}\text{Comp(P)} = \{\quad & \forall x(\text{koululainen}(x) \leftrightarrow x = \text{oskari}),\\ & \forall x(\text{sairas}(x) \leftrightarrow x = \text{liisa}),\\ & \forall x(\text{koulussa}(x) \leftrightarrow (\neg\text{sairas}(x) \wedge \text{koululainen}(x))),\\ & \neg(\text{oskari} = \text{liisa})\}\end{aligned}$$

Mutta: Comp(P) $\models \exists x \text{koulussa}(x)$.

3 Sääntöpohjainen päättely

Monissa järjestelmissä (esim. logiikkaohjelmointi, deduktiiviset tietokannat, asiantuntijajärjestelmät) voidaan käyttää muotoa $A \leftarrow A_1, \ldots, A_n$ olevia sääntöjä.

- Ideana on, että säännön **seuraus** A voidaan päätellä, kunhan ensin säännön **ehdot** A_1, ..., A_n on saatu pääteltyä.
- Sääntöä kutsutaan *ehdottomaksi*, jos $n = 0$.
- Tämä sääntö voidaan esittää lauselogiikan implikaationa $A_1 \wedge \ldots \wedge A_n \rightarrow A$, joka on normaalimuodossaan $\neg A_1 \vee \ldots \vee \neg A_n \vee A$ Horn-klausuuli.

Rajoitutaan jatkossa sääntöihin, joissa esintyy ainoastaan atomilauseita (ts. lauselogiikan tapaukseen).

Väite. Sääntöjoukon S säännöillä voidaan päätellä atominen lause A, jos ja vain jos A on looginen seuraus S:n sääntöjä $A \leftarrow A_1, \ldots, A_n$ vastaavista implikaatioista $A_1 \wedge \ldots \wedge A_n \rightarrow A$.

Merkitään jatkossa säännöillä S pääteltävissä olevien atomisten lauseiden joukkoa merkinnällä $\mathrm{Cn}(S)$.

Ratkaistava ongelma: voidaanko säänntöjoukon S säännöillä päätellä annettu atominen lause A? Entä jos sääntöjoukkoon S lisätään aluksi joukko ehdottomia sääntöjä F (lähtötiedot)?

Tehokas algoritmi näihin ongelmiin: Dowling-Gallier [1984]

Tietorakenteet

Otetaan käyttöön seuraavat taulukot.

- queue: jono atomeista A, jotka esiintyvät seurauksena sääntöjoukon ehdottomissa säännössä.
- val[A]: kuuluuko atomi A joukkoon Cn(S)?
  Aluksi true, jos A on jonossa queue, ja false muutoin.
- rules[A]: lista säännöistä, joissa atomi A esiintyy ehtona.
- counter[s]: alussa säännön s ehtojen lukumäärä
- head[s]: säännön s seuraus

Näiden laskennan aikavaativuus on luokkaa $\mathcal{O}(n \times \log n)$.

Lineaarinen algoritmi

```
function cn:  while not empty(queue) do
                A := get(queue);  rlist := rules[A] ;
                while not empty(rlist) do
                  rule := first(rlist); rlist := rest(rlist);
                  counter[rule] := counter[rule] - 1;
                  if counter[rule] = 0 then
                    B:= head[rule];
                    if val[B] = false then
                      val[B] := true;
                      put(B, queue)
                    endif
                  endif
                endwhile
              endwhile
```

Esimerkki. $S =$
$\{1 : P_0$
$2 : P_1 \leftarrow P_0$
$3 : P_2 \leftarrow P_1, P_0$
$4 : P_3 \leftarrow P_1$
$5 : P_4 \leftarrow P_1, P_4\}$

Lopputila :

val[0]	true
val[1]	true
val[2]	true
val[3]	true
val[4]	false

val[0] := true
queue := $\langle 0 \rangle$
A := 0
rules := [2,3]
  rule := 2
  counter[2] := 0
  val[1] := true
  queue := $\langle 1 \rangle$
  rule := 3
  counter[3] := 1
A := 1
rules := [3,4,5]
  rule := 3
  counter[3] := 0
  val[2] := true
  queue := $\langle 2 \rangle$
  rule := 4
  counter[4] := 0
  val[3] := true
  queue := $\langle 2, 3 \rangle$
  rule := 5
  counter[5] := 1
A := 2
rules := []
A := 3
rules := []

4 Induktioperiaateen käyttökohteita

Määritelmä. (Induktioperiaate luonnollisille luvuille)
Jos $P(0)$ ja $P(n) \rightarrow P(n+1)$ kaikille luonnollisille luvuille n,
niin $P(n)$ kaikille luonnollisille luvuille n.

Monelle induktioperiaate on tuttu matematiikasta:

Esimerkki: Todistetaan, että $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$.

Perustapaus: $2^0 = 1$ ja $2^1 - 1 = 2 - 1 = 1$.

Induktiohypoteesi: $2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$

Induktioaskel:
$$\begin{aligned} 2^0 + 2^1 + \ldots + 2^n &= (2^0 + 2^1 + \ldots + 2^{n-1}) + 2^n \\ &= (2^n - 1) + 2^n \\ &= 2 \times 2^n - 1 = 2^{n+1} - 1 \end{aligned}$$

Toistolausekkeiden invariantit

Tietojenkäsittelyssä induktiota voidaan käyttää mm. ohjelmien oikeellisuuden toteamisessa. Tarkastellaan seuraavaa esimerkkiä:

```
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;
}
```


Todistetaan induktiolla silmukan suorituskertojen (k) suhteen, että:

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

missä s_k ja i_k ovat `s`:n ja `i`:n arvot kerralla k.

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

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

Induktiohypoteesi (tilanne k:n suorituskerran jälkeen):

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

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

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

Todistamamme ominaisuus on esimerkki silmukan *invariantista*, eli ominaisuudesta, joka pysyy voimassa suorituskerrasta toiseen.

Silmukan suoritus päättyy, kun $i_k < n$ tulee epätodeksi.
Koska tällöin $i_k \leq n$ on edelleen tosi, tiedämme, että $i_k = n$.

Tällöin muuttujan `s` arvona on

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

mikä onkin taulukon `a` lukujen summa.

Rakenteinen induktio

Induktiota voidaan käyttää myös induktiivisesti määriteltyjä tietorakenteita käsittelevien ohjelmien analysoinnissa.

Esimerkkinä olkoon proseduuri `reverse`, joka kääntää listan jäsenet käänteiseen järjestykseen (listasta `[1,2,3]` saadaan `[3,2,1]`).

Oikeellisuustarkastelussa osoitetaan induktiolla, että `reverse` käsittelee pituudeltaan n olevat listat oikein.

Analysoidaan seuraavaa Prolog-toteutusta, jossa on käytetty apuproseduuria/apupredikaattia `append` (jonka oikeellisuuden joudumme osoittamaan samalla).

```
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).
```

Todistettava ominaisuus:

Prolog-tulkki vastaa "kyllä" (jos tulkki pystyy johtamaan kyselystä tyhjän klausuulin) antaen vastaussubstituution θ kyselyyn

- `:-reverse(L,M)` **joss** lista `M`θ on lista `L` käännettynä
- `:-append(L,A,M)` **joss** listassa `M`θ esiintyvät ensin listan `L` alkiot ja sitten alkio `A`

missä `L`, `M`θ ja `A` ovat muuttujattomia ja listan `L` pituus on n.

Osoitetaan induktiolla, että kaikilla luonnollisilla luvuilla n on em. ominaisuus (joten `reverse` ja `append` käsittelevät oikein mielivaltaisen pitkiä listoja).

Perustapaus ($n = 0$):

Koska $n = 0$, listan `L` täytyy olla tyhjä lista `[]`.

Tällöin kysely `:-reverse([],M)` palauttaa vastauksen θ
joss `M`θ=`[]` (θ on termien `M` ja `[]` MGU)
joss lista `M`θ on lista `[]` käännettynä.

Vastaavasti kysely `:-append([],A,M)` palauttaa vastauksen θ
joss `M`θ=`[A]` (θ on termien `M` ja `[A]` MGU)
joss listassa `M`θ esiintyy ensin listan `[]` alkiot ja sitten alkio `A`.

Induktiohypoteesi:
todistettava ominaisuus on voimassa, kun listan `L` pituus on n.

Induktioaskel: Olkoon listan `L1` pituus $n+1$.

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

joss θ_1 on atomien `reverse(L,M)` ja `reverse([X|Y],Z)` MGU,
`L`=`[X|Y]`θ_1=`[X`θ_1`|Y`θ_1`]` ja `M`θ_1=`Z`θ_1,
kysely `:-reverse(Y`θ_1`,V)` palauttaa vastauksen θ_2 ja
kysely `:-append(V`θ_2`,X`θ_1`,Z`θ_1`)` palauttaa vastauksen θ_3

joss θ_1 on atomien `reverse(L,M)` ja `reverse([X|Y],Z)` MGU,
`L`=`[X|Y]`θ_1=`[X`θ_1`|Y`θ_1`]` ja `M`θ_1=`Z`θ_1,
lista `V`θ_2 on lista `Y`θ_1 käännettynä ja listassa
`Z`$\theta_1\theta_3$ esiintyy ensin listan `V`θ_2 alkiot ja sitten alkio `X`θ_1

joss lista `M`θ= `M`$\theta_1\theta_3$=`Z`$\theta_1\theta_3$ on lista `L`=`[X`θ_1`|Y`θ_1`]` käännettynä.

Huom! Yllä `L`, `X`θ_1, `Y`θ_1, `V`θ_2 ja `Z`$\theta_1\theta_3$ ovat muuttujattomia.

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

joss θ_1 on atomien `append(L,A,M)`
ja `append([X|Y],Z,[X|V])` MGU,
`L`=`[X|Y]`θ_1=`[X`θ_1`|Y`θ_1`]`, `A`=`Z`θ_1 ja `M`θ_1=`[X|V]`θ_1= `[X`θ_1`|V`θ_1`]`
ja kysely `:-append(Y`θ_1`,Z`θ_1`,V`θ_1`)` palauttaa vastauksen θ_2

joss θ_1 on atomien `append(L,A,M)`
ja `append([X|Y],Z,[X|V])` MGU,
`L`=`[X|Y]`θ_1=`[X`θ_1`|Y`θ_1`]`, `A`=`Z`θ_1 ja `M`θ_1=`[X|V]`θ_1= `[X`θ_1`|V`θ_1`]`
listassa `V`$\theta_1\theta_2$ esiintyy listan `Y`θ_1 alkioiden jälkeen alkio `Z`θ_1.

joss listassa `M`θ= `M`$\theta_1\theta_2$= `[X`θ_1`|V`$\theta_1\theta_2$`]` esiintyy ensin listan
`L`=`[X`θ_1`|Y`θ_1`]` alkiot ja sitten alkio `Z`θ_1=`A`.

Huom! Yllä `L`, `A`, `X`θ_1, `Y`θ_1, `Z`θ_1, `V`$\theta_1\theta_2$ ovat muuttujattomia.

Haluatko tietää logiikasta lisää ???

Tietojenkäsittelyteorian laboratorion kurssitarjonnassa on mm.

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

Näillä käsitellään lisää logiikkaan ja laskennalliseen vaativuuteen liittyviä aiheita.