IV.5.3 Todistussemantiikka

Todistusmenetelmän pitää olla **oikeellinen**: sillä voi todistaa *vain varmoja asioita*.

Varmat asiat ovat ne, jotka ovat *tosia kaikissa malleissa*.

Kalvojen IV.5.2 mukaan Prolog-ohjelmalla L on *pienin* malli M_L joka on samalla sen merkitys.

Kysymykseen "Onko maalifakta $p \in M_L$?" – eli kalvojen IV.5.1 aliohjelmakutsuun p – voi siis yrittää vastata *todistamalla* p väitteistä L.

Valitaan *täydellinen todistussääntö* eli sellainen jolla pystyy todistamaan kaiken sen, mikä on varmaa.

Kalvojen IV.5.2 *todistuksen etsinnän* epätäydellisyyteen se ei auta: yleisessä tapauksessa on mahdotonta tunnistaa millään algoritmilla, että todistusta *ei* ole.

Prolog-ohjelmien todistussäännön perusaskeleer käytetään *resoluutiota*. Sen muuttujaton muoto

looginen ehto 1: $q_1 \wedge \ldots \wedge q_m \rightarrow p_1 \vee \ldots \vee p_n$

looginen ehto 2: $q'_1 \wedge \ldots \wedge q'_{m'} \rightarrow p'_1 \vee \ldots \vee p'_{n'}$

sivuehto: p_1 ja q'_1 ovat samat

johtopäätös:
$q_1 \wedge \ldots \wedge q_m \wedge q'_2 \wedge \ldots \wedge q'_{m'} \rightarrow p_2 \vee \ldots \vee p_n \vee p'_1 \vee \ldots$

missä kaikki p:t ja q:t ovat faktoja.

"Jos yhden ehdon oikealla puolella on sama kui toisen vasemmalla, niin ne voidaan pyyhkiä pois loput yhdistää uudeksi ehdoksi."

Muuttujat mukaan resoluutiosääntöön:

- Kummankin loogisen ehdon edessä on "kaikilla"-kvanttori $\forall \vec{X}$, $\forall \vec{X}'$ jokaiselle ehdon muuttujalle, kuten kalvoilla IV.5.2.

- Nimetään loogisten ehtojen muuttujat uude siten, ettei sama nimi esiinny molemmissa: $\forall \vec{Y}'$.

- Luetaan sivuehto "... samastuvat yleisimmällä samastimella θ".

- Johtopäätökseen sovelletaan tätä samastinta θ.

- Johtopäätöksen edessä on kvanttori $\forall \vec{Z}$ jokaiselle siihen jääneelle muuttujalle.

Miten nämä implikaatiot liittyvät Prologiin?

- Kalvoilla IV.5.2 sääntö

$$p\texttt{:-}q_1\texttt{,...,}q_m\texttt{.}$$

tulkittiin loogisesti

$$\forall \vec{X}.q_1 \wedge \ldots \wedge q_m \rightarrow p.$$

- Koska tyhjä konjunktio ("ja") on **tosi**, niin puuttuva vartalo (eli $m = 0$) eli fakta

$$p\texttt{.}$$

on

$$\forall \vec{X}.\textbf{tosi} \rightarrow p$$

eli

$$\forall \vec{X}.p.$$

- Koska tyhjä disjunktio ("tai") on **epätosi**, n puuttuva pää on

$$\forall \vec{X}.q_1 \wedge \ldots \wedge q_m \rightarrow \textbf{epätosi}$$

eli

$$\forall \vec{X}.\neg(q_1 \wedge \ldots \wedge q_m)$$

(missä '$\neg$' on negaatio eli "ei"), eli

$$\neg\exists \vec{X}.q_1 \wedge \ldots \wedge q_m$$

(missä '$\exists$' on "on olemassa" -kvanttori).

"Ei ole olemassa sellaisia arvoja muuttujille joilla ongelmat $q_1, \ldots, q_m$ ratkeaisivat yhtä ai

Tämä on *maali*

$$\texttt{?-}q_1\texttt{,...,}q_m\texttt{.}$$

jota yritetään *kumota* vastauksella "onpas; esimerkiksi arvot...".

Resoluutio onkin *refutaatio*- eli ristiriitatäydellinen päättelysääntö: jos säänn maali eivät voi olla yhtä aikaa totta, niin se osoittaa resoluutioaskeleilla.

Prologissa resoluutioaskel saa siis muodon

sääntö: p:-q_1,...,q_m.

maali: ?-q'_1,...,$q'_{m'}$.

sivuehto: säännön päällä p ja ensimmäisellä alimaalilla q'_1 on yleisin samastin θ.

Ensin säännön muuttujat on kuitenkin nimetty erilleen maalin muuttujista.

uusi maali: ?-$q_1\theta$,...,$q_m\theta$,$q'_2\theta$,...,$q'_{m'}\theta$.

Maali toimii siis (ensimmäistä) ratkaisuaan odottavien alimaalien *pinona*.

Kun saavutetaan *tyhjä* maali ?-., on löydetty ristiriita **tosi** $\rightarrow$ **epätosi**. Vastaesimerkkinä kerrotaan todistuksen kuluessa kertynyt sidonta. Kun vielä päätetään

- kokeilla sääntöjä niiden kirjoitusjärjestyksessä
- peruuttaa edelliseen maaliin ja sen seuraavaan sääntöön kun nykyinen maali osoittautuu mahdottomaksi

päästään kalvojen IV.5.1 suorituskoneeseen.


```
pajakari$ /opt/pl/bin/pl
Welcome to SWI-Prolog (Version 4.0.1)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('sum.pl').
% sum.pl compiled 0.00 sec, 1,112 bytes

Yes
?- trace,p(X,Y,s(s(s(o)))).
   Call: (7) p(_G282, _G283, s(s(s(o)))) ? creep
   Exit: (7) p(o, s(s(s(o))), s(s(s(o)))) ? creep

X = o
Y = s(s(s(o))) ;
   Redo: (7) p(_G282, _G283, s(s(s(o)))) ? creep
   Call: (8) p(_G378, _G283, s(s(o))) ? creep
   Exit: (8) p(o, s(s(o)), s(s(o))) ? creep
   Exit: (7) p(s(o), s(s(o)), s(s(s(o)))) ? creep

X = s(o)
Y = s(s(o)) ;
   Redo: (8) p(_G378, _G283, s(s(o))) ? creep
   Call: (9) p(_G380, _G283, s(o)) ? creep
   Exit: (9) p(o, s(o), s(o)) ? creep
   Exit: (8) p(s(o), s(o), s(s(o))) ? creep
   Exit: (7) p(s(s(o)), s(o), s(s(s(o)))) ? creep

X = s(s(o))
Y = s(o) ;
   Redo: (9) p(_G380, _G283, s(o)) ? creep
   Call: (10) p(_G382, _G283, o) ? creep
   Exit: (10) p(o, o, o) ? creep
   Exit: (9) p(s(o), o, s(o)) ? creep
   Exit: (8) p(s(s(o)), o, s(s(o))) ? creep
   Exit: (7) p(s(s(s(o))), o, s(s(s(o)))) ? creep

X = s(s(s(o)))
Y = o ;
   Redo: (10) p(_G382, _G283, o) ? creep
   Fail: (10) p(_G382, _G283, o) ? creep
   Fail: (9) p(_G380, _G283, s(o)) ? creep
   Fail: (8) p(_G378, _G283, s(s(o))) ? creep
   Fail: (7) p(_G282, _G283, s(s(s(o)))) ? creep

No
[debug]  ?-
```

Edellä todistussemantiikka kuvattiin samassa järjestyksessä kuin kalvojen IV.5.1 suoritussemantiikassa, jossa sääntöjä käytetään "takaperin" eli **niin**-osasta **jos**-osaan.

Kalvojen IV.5.2 mallisemantiikassa sääntöjä käytettiinkin "etuperin". Vastaava todistussemantiikka saadaan lukemalla Prolog-lause

$$p(t_{(i,1)},\ldots,t_{(i,k)}) \text{ :- } q_{(i,1)},\ldots,q_{(i,m_i)}. \text{ eli}$$
$$\forall\vec{X}.q_{(i,1)} \wedge \ldots \wedge q_{(i,n)} \rightarrow p(t_{(i,1)},\ldots,t_{(i,k)})$$

(ei enää totuudesta puhuvana väitteenä vaan) todistuvuudesta puhuvana päättelysääntönä

$$\frac{q_{(i,1)} \quad \cdots \quad q_{(i,m_1)}}{p(t_{(i,1)},\ldots,t_{(i,k)})}$$

joka luetaan "jos olet saanut todistuksen K_i jokaiselle oletukselle $q_{(i,j)}$, niin voit yhdistää ne todistukseksi K johtopäätökselle $p(t_{(i,1)},\ldots,t_{(i,k)})$".

Muuttujat tulkitaan kalvojen IV.5.1 hahmojen tavoin:

- Jokainen K_i todistaa jokin muuttujattoman muodon q_i' omasta johtopäätöksestään $q_{(i,j)}$.
- Näiden q_i' on yhdessä sovittava säännön hahmoon.
- Sovittamisella määräytyy johtopäätöksen todistettu muuttujaton (?) muoto.

Tässä ohjelman luennassa päättelysääntöinä kal
IV.5.1 suoritussemantiikka voidaan selittää seuraavasti:

- Todistusta etsitäänkin "alhaalta ylöspäin" e johtopäätöksistä oletuksiin.
- Etsinnän kuluessa muuttujia sidotaan samastuksella väin sen verran kuin on välttämätöntä kokeiltavan säännön käyttämiseksi.
- Sääntöjä kokeillaan kirjoitusjärjestyksessä $i = 1,\ldots,n$.
- Oletukset yritetään todistaa kirjoitusjärjestyksessä $j = 1,\ldots,m_i$.

Entä mikä on tässä luennassa sellainen looginer päättelysäännöstö, jossa annetun väitteen todis etsittäessä ei koskaan ole kuin olennaisesti yksi mahdollisuus?

Sehän on ohjelmointikielen suoritusmekanismin abstrakti määrittely "vaihtoehdottomana logiikkana"! Esimerkkinä kalvojen II.5 Scheme-ydinkielen määrittely päättelysäännöillä

Vastaava malliteoria on kielen denotationaaliner tarkoitesemantiikka.

IV.6 Negaatio tarkemmin

Kalvojen IV.5 puhtaassa Prologissa ei voi ilmaista *negatiivista* tietämystä "... *ei* ole...".

- Olisi yksinkertaista kirjoittaa faktoja "henkilö X sairastaa tautia Y" ja
- määritellä "henkilö Z on terve jos hän ei sairasta mitään tautia".

Tämän vuoksi Prologiin on lisätty kalvojen IV.4.4 negaatio epäonnistumisena [B01,§5.3-5.4]: vartalossa voi esiintyä alimaali

$$\texttt{not}(p)$$

jonka haluttu luenta on "väite p ei päde".

Seuraavaksi tarkastellaan mikä on sen todellinen luenta kalvojen IV.5.1—IV.5.3 näkökulmista.

IV.6.1 Suoritussemantiikassa

Kalvojen IV.5.1 suoritussemantiikassa kutsu `not` tehdään seuraavasti:

1. Yritetään suorittaa negaation sisällä oleva k p nykyisellä sidonnalla θ_i.

2. Jos sisäkutsu p epäonnistuu, niin koko kutsu `not`(p) *onnistuu*.

   Sidontana pysyy θ_i.

   (Eihän epäonnistunut p voinut kasvattaa sid θ_i. Myöskään myöhempi peruutus takaisin kutsuun `not`(p) ei tuota uusia vaihtoehtoja.)

3. Muuten koko kutsu *epäonnistui* (ja peruutet hakemaan vaihtoehtoista θ_i).

Sairausesimerkkimme on nyt käytännössä:

```
sairastaa(leopold_duke_of_albany,hemofilia).
sairastaa(rupert,hemofilia).
sairastaa(frederick,hemofilia).
sairastaa(alexis,hemofilia).
sairastaa(waldemar,hemofilia).
sairastaa(henry,hemofilia).
sairastaa(leopold,hemofilia).
sairastaa(maurice,hemofilia).
sairastaa(alfonso,hemofilia).
sairastaa(gonzalo,hemofilia).
sairastaa(luennoija,kevätflunssa).

terve(X):-not(sairastaa(X,_)).

?- trace,terve(luennoija).
   Call: (7) terve(luennoija) ? creep
^  Call: (8) not(sairastaa(luennoija, _G319)) ? creep
   Call: (9) sairastaa(luennoija, _G319) ? creep
   Exit: (9) sairastaa(luennoija, kevätflunssa) ? creep
^  Fail: (8) not(sairastaa(luennoija, _G319)) ? creep
   Fail: (7) terve(luennoija) ? creep

No

?- trace,terve(teräsmies).
   Call: (7) terve(teräsmies) ? creep
^  Call: (8) not(sairastaa(teräsmies, _G313)) ? creep
   Call: (9) sairastaa(teräsmies, _G313) ? creep
   Fail: (9) sairastaa(teräsmies, _G313) ? creep
^  Exit: (8) not(sairastaa(teräsmies, _G313)) ? creep
   Exit: (7) terve(teräsmies) ? creep

Yes
```

Tähän saakka vastaukset olivat järkeviä, mutta...

```
?- trace,terve(X).
   Call: (8) terve(_G225) ? creep
^  Call: (9) not(sairastaa(_G225, _G284)) ? creep
   Call: (10) sairastaa(_G225, _G284) ? creep
   Exit: (10) sairastaa(leopold_duke_of_albany, hemofilia) ? creep
^  Fail: (9) not(sairastaa(_G225, _G284)) ? creep
   Fail: (8) terve(_G225) ? creep

No
```

Kalvoilla IV.5.3 Prolog-maali `?-` q`.` luettiin "onk
olemassa sellaisia vapaiden muuttujien arvoja, jo
pätee (ja jos on niin mitkä ne ovat)?"

Tarkoitimme maalillamme `?- terve(X).` siis "ha
kaikki terveet ihmiset `X`".

Saimme vastauksen `No` eli "heitä ei ole":

1. Normaali suoritus johti ensin uuteen maaliin
   `?- not(sairastaa(X,_)).`

2. Valittu negaation suoritustapa teki siitä maa
   `?- sairastaa(X,_).`

3. Tämän maalin `Yes/No`-vastaus käännetään n

   Mutta tämä maalihan luetaankin "sairastaal
   joku jotakin?"

   Siis alkuperäinen maali kysyykin "ovathan k
   terveitä?"

Tämän vuoksi negaatioon epäonnistumisena ei
luottaa jos negatoidussa alimaalissa on kutsuhe
vapaita muuttujia.

Maalin implisiittinen $\exists$`X.terve(X)` vaihtuu näet
negaation ajaksi muotoon $\forall$`X,_.`$\neg$`sairastaa(X,_)`

IV.6.2 Mallisemantiikassa

Kalvojen IV.6.1 negatoitujen muuttujien ongelmaa voi yrittää selittää kalvojen IV.5.2 mallisemantiikalla.

Positiivinen muuttujaton maali `?-` p`.` vastaa `Yes/No`-kysymykseen "$p \in M_L$".

Negatiivinen muuttujaton `?- not(`q`).` vastaa kysymykseen "$q \notin M_L$".

Tämän toteuttu negaatio kyllä osaa.

Positiivinen muuttujallinen p on hahmo joka edustaa kaikkia niitä muuttujattomia p' joilla vastaus on `Yes`.

Negatiivinen muuttujallinen q edustaa kaikkia q' joilla vastaus on `No`.

Tätä toteutettu negaatio ei osaa: se tarkistaa vain onko sellaisia olemassa.

HUOM! Tämä selittää vain käyttäjän *kyselyihin* kirjoittamat negaatiot. *Ohjelmaan* kirjoitetut negaatiot saattavat (rekursioon osallistuessaan) jopa tuhota pienimmän mallin M_L yksikäsitteisyyden!

IV.6.3 Todistussemantiikassa

Kalvojen IV.5.3 todistussemantiikassa kalvoilla I valittu negaation toteutus on uusi päättelysään

Väite `not(`p`)` voidaan todistaa osoittamalla ettei väitettä p voi todistaa.

Tämä on ns. *suljetun maailman oletus* (Closed World Assumption, CWA). Sitä käytetään paljo tekoälyssä.

Koska todistaa sai vain sellaista mikä on varma tämä oletus (täydellisessä päättelyjärjestelmäss

"koska p ei ole varmaa niin se ei ole totta".

Ohjelman "maailma(nkuva)" sulkee siis pois *epävarmuuden*: jos ohjelmalla ei ole tarpeeksi ti väitteestä p, niin se ei ohjelman mielestä voi pit paikkaansa.

Tietojenkäsittelyllisesti perusteltavissa: ohjelma reagoitava oman tietämyksensä pohjalta vaikke tiedäkään kaikkea ulkomaailmasta.

Tämä on ns. *epämonotoninen* päättelysääntö: u tiedon lisääminen saattaa *kumota* aiempia pääte

IV.7 Haun ohjailu

Kalvojen IV.4.4 Prolog-negaatio toteutettiin kalvoilla IV.4.5 ohjailemalla Prolog-hakukoneen toiminta kirjastopredikaateilla `!/0` ja `fail/0`.

Tarkastellaan katkaisupredikaattia `!/0` tarkemmin. Olkoon ratkaistavana maali

$$\texttt{?-}g\texttt{,...}$$

ja kokeiluvuorossa ohjelman sääntö

$$p\texttt{:-}q_1\texttt{,...,}q_m\texttt{,!,}r_1\texttt{,...,}r_n\texttt{.}$$

joka vaikuttaa sopivalta katkaisuun saakka, eli on saatu sidonta θ_m (samastamalla g ja p sidonnaksi θ_0, laajentamalla se kutsulla q_1 sidonnaksi θ_1,...).

- Suoritus etenee normaalisti eteenpäin yli katkaisukohdan vartalon loppuosaan $r_1,\ldots,r_n$. sidonnan θ_m muuttumatta.

- Sivuvaikutuksena etsintäkone hävittää kaikk kutsun g jäljellä olleet vaihtoehtoiset ratkaisutavat:

  – vartalon alkuosan $q_1,\ldots,q_m$

  – loput ohjelmasta.

- Jos suoritus myöhemmin peruuttaa taaksepä katkaisukohdan, niin koko nykyinen maali epäonnistuu heti; sen ensimmäistä kutsua g voida enää ratkaista toisin.

  Suoritus jatkuu eteenpäin edellisestä maalist siitä jossa g ei vielä ollut ensimmäisenä.

Kalvojen IV.4.5 negaation toteutus on esimerkk säännöstä (uutta) muotoa

jos...niin ...<u>muuten</u>...

"**Jos** `P` onnistuu **niin** epäonnistu **muuten** onnis

Tähän on tarjolla syntaktista sokeria muodossa `P -> fail ; true.`

Tässä väliin kirjoitettu funktori `->/2` on **jos...niin...**.

Väliin kirjoitettu funktori `;/2` taas on **tai**.

\+ Katkaisu laajentaa Prolog-kielen ilmaisuvoimaa.

\+ Katkaisujen lisääminen parantaa Prolog-ohjelman tehokkuutta koska turha peruutus jää pois.

– Katkaisun käyttö tekee Prolog-ohjelmasta (lopullisesti) sellaisen jonka sääntöjä on seurattava kirjoitusjärjestyksessä.

Katkaisu on

vihreä jos se tehostaa ohjelmaa muuttamatta sen merkitystä (= löytämiä vastauksia)

punainen jos ohjelman merkitys muuttuu.

Punaisia katkaisuja tulee käyttää varoen, koska deklaratiiviselta näyttävään ohjelmaan on nyt ilmaantunut proseduraalisia riippuvuuksia!

Kalvojen IV.3 2-3-puuesimerkkiin voidaan lisätä vihreitä katkaisuja koska sen säännöt ovat (vertailujen nojalla) toisensa poissulkevia:

```
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        X<Y,
        !,
        ins23(T0,X,TX),
        put3l(TX,Y,TY,Z,TZ,T).
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        Y<X,
        X<Z,
        !,
        ins23(TY,X,TX),
        put3m(T0,Y,TX,Z,TZ,T).
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        Z<X,
        ins23(TZ,X,TX),
        put3r(T0,Y,TY,Z,TX,T).
```

Mutta jos koodia optimoi edelleen poistamalla *toisesta* säännöstä (nyt tarpeettoman) vertailun niin *ensimmäisen* säännön katkaisusta tuleekin punainen:

- sääntöjä ei voi enää järjestellä vapaasti uude
- ensimmäisen säännön katkaisua ei saa enää poistaa.

Tässä punainen katkaisu ei voittane paljoakaan.

On myös perusteltuja punaisia katkaisuja: samassa esimerkissä säännöt

```
put2l(c(T0,X,TX),Y,TY,
    t(  T0,X,TX, Y,TY)).
put2l(b(T0,X,TX),Y,TY,
    b(b(T0,X,TX),Y,TY)).
put2l(t(T0,X,TX,Y,TY),Z,TZ,
    b(t(T0,X,TX,Y,TY),Z,TZ)).
```

sanovat "jos vasen alipuu on kasvanut korkeutta, niin tee asialle jotakin; muuten se käy sellaisenaan".

Tämä voidaan ilmaista luontevasti, lyhesti ja tehokkaasti

```
put2l(c(T0,X,TX),Y,TY,
    t(  T0,X,TX, Y,TY)):-
        !.
put2l(T,Y,TY,b(T,Y,TY)).
```

punaisella katkaisulla.

Punaisia katkaisuja käytettäessä on syytä noudattaa erityistä varovaisuutta ja huolellisuutta. . .

Toinen tapa ohjailla hakua on kirjoittaa sellaisia Prolog-ohjelmia, joissa sääntöjä valitaan sen mu millaista *tyyppiä* eri muuttujien arvot nyt ovat.

Prolog on samaan tapaan piilevästi tyypitetty k kuin Scheme (kalvot II.4).

Tyyppien tutkimiseen on seuraavat kirjastopredikaatit [B01,7.1]:

`atom`(X) kertoo, onko X jakamaton *atomi* — Prolog-vastine kalvojen II.4.3 Scheme-symboleille.

Kalvojen IV.4.1 tyhjä Prolog-lista `[]` on my atomi.

`integer`(X) kertoo, onko X kokonaisluku (kalvot IV.4.2).

`float`(X) kertoo, onko X liukuluku.

`number`(X) kertoo, onko X kokonais- tai liukulu

`atomic`(X) kertoo, onko X atomi tai luku.

`compound`(X) kertoo, onko X rakenteinen termi, eli muotoa $f(\ldots)$.

Myös kalvojen IV.4.1 epätyhjät Prolog-listat ovat rakenteisia. Niissä funktorina f on sama '.'/2 joka nähtiin jo kalvojen II.8 Scheme-listoissa.

`var`(X) kertoo, onko X tällä hetkellä sellainen muuttuja, jota ei vielä ole sidottu.

Jos X on sidottu, mutta vain johonkin toiseen vapaaseen muuttujaan Y, niin tätä yhteyttä "X ja Y ovat vapaat vapaat mutta samat" ei vielä pidetä sidontana.

Näin `var`(X) kertoo esimerkiksi, voiko X toimia tulospositiona — jolloin se ei voi olla syötepositio...

`nonvar`(X) kertoo käänteisen ominaisuuden: joko X ei ole muuttuja tai sitten se on sellainen muuttuja, joka on tällä hetkellä sidottu johonkin "oikeaan" arvoon, eli atomiin, numeroon tai rakenteiseen termiin.

IV.8 Prolog ja tietokannat

Kalvojen III yksinkertaisessa Prolog-tulkissa sää ja faktat talletettiin alkeelliseen "tietokantaan" listaan kirjoitusjärjestyksessä.

Myös oikeaan Prolog-toteutukseen kuuluu *sisäi tietokanta* jossa sääntöjä säilytetään – edelleen järjestyksessä siltä varalta että ohjelman merkit riippuisi sääntöjen suoritusjärjestyksestä...

Prologilla on myös yhteyksiä ulkoisten *relaatiotietokantojen* moderneihin kyselykieliin.

Lisäksi useissa Prolog-toteutuksissa on mahdoll käyttää jotakin ulkoista relaatiotietokantaa muuttujattomien faktojen tallennukseen. Esime SWI Prolog tarjoaa Open DataBase Connectivi (ODBC) -standardin mukaisen kirjaston.

Prolog on siis hyvä korkean tason kieli sellaisiin tiedonhallintatehtäviin, joissa tarvitaan relaatio-operaatioiden lisäksi päättelykykyä.

IV.8.1 Sisäinen tietokanta

[B01,§7.4]

`asserta(C)` (onnistuu aina kasvattamatta sidontaa ja) lisää sisäiseen tietokantaan säännön `C` muiden saman predikaatin sääntöjen *eteen*.

(Jos `C` on vartalollinen, niin se on syntaktisista syistä kirjoitettava toisiin sulkuihin).

Peruutettaessa `asserta/1` ei anna uusia "vastauksia", mutta sen lisäämä `C` jää tietokantaan.

`assertz(C)` on kuin `asserta/1` paitsi että `C` lisätäänkin edellisten *perään*.

`assert(C)` on näistä kirjastopredikaateista se jota suositellaan käytettäväksi kun sisäisen tietokannan järjestyksellä ei ole väliä – esimerkiksi kun lisätään (muuttujaton) fakta eli talletetaan tietoalkioita eikä aliohjelmia.

(Yleensä `assert/1` on `assertz/1`.)

`retract(C)` poistaa sisäisestä tietokannasta ensimmäisen säännön (tai faktan) joka sop hahmoon `C` ja laajentaa sidontaa siten kuin sopii poistettuun sääntöön.

Peruutettaessa poistetaan myös toinen, kolmas,. . . kunnes poistettavaa ei enää ole.

Esimerkiksi maali

```
?- retract((apu(X):-B)).
```

poistaa predikaatin `apu/1` ensimmäisen sell säännön r jolla on vartalo sekä sitoo muut `X` säännön r pään parametriin ja muuttujan säännön r vartaloon.

Maali

```
?- retract(apu(Y)),fail.
```

taas poistaa peruutellen kaikki predikaatin vartalottomat faktat.

Tässä on *toisto epäonnistuen* (failure-drive loop): tietokannalla on *tila joka muuttuu sivuvaikutuksena*, joten `fail`-epäonnistumis aiheuttama peruutus toistaa tätä sivuvaikutusta.

Myös kalvojen IV.4.6 tulostus tapahtui sivuvaikutuksin.

Prolog-standardissa (mutta ei välttämättä kaikissa toteutuksissa)

- `consult`oitu tai kirjastopredikaatti on *staattinen* eikä muokattavissa suoritusaikana ilman edeltävää kutsua `dynamic`/1.

  Intuitiivisesti, suorituskelpoinen ohjelma*koodi* pidetään sittenkin erillään vapaasti muokattavasta tietokanta*datasta*, kunnes erikseen julistetaan aikomus muokata jotakin osaa koodista.

- vaikka predikaatin määrittelyä muutettaisiinkin kesken sen suorituksen, jatkuu sen nykyinen suoritus loppuun sillä samalla määrittelyllä jolla alkoikin [DEC96,§10.2].

  Intuitiivisesti, tietokantapäivitykset *ensin laskevat* vanhan tietokannan sisällöstä sen, millainen sisältö uudelle tietokannalle halutaan, ja vasta *sitten vaihtavat* vanhan sisällön uuteen.

  (Sen epäloogisempi mutta usein käytetty vaihtoehto on, että jokainen yksittäinen `assertz`/1 ja `retract`/1 tehdään heti, ja sen tulos vaikuttaa jo saman päivityksen seuraaviin askeliin.)

IV.8.2 Datalog

Kalvojen IV.8.1 ajatusta datatietokannan pitäm erillään säännöistä voi jatkaa edelleen:

- Unohdetaan rakenteiset termit (kuten listat) tyydytään pelkkiin atomeihin.

- **Ekstensionaalinen** tietokanta koostuu pelkästään muuttujattomista – eli vain aton sisältävistä – faktoista.

  Kukin tällainen predikaatti vastaa tutun *relaatiomallin* [AHV95,§3] yhtä (muokattava tietokantataulua.

  ```
  töissä(työntekijä,yritys)
  johtaa(johtaja,yritys)
  valvoo(konserniyritys,tytäryritys)
  ```

- **Intensionaalinen** tietokanta sisältää ne sää joilla ekstensionaalista kysellään: kyselyt, näkymät,...

  Säännöissä on eri predikaatit kuin tauluissa.

Päädytään kyselykieleen *datalog* (database Pro [AHV95,§D].

Intensionaaliseen kantaan voidaan kirjoittaa relaatiokyselyjä kuten "pomot ja alaiset":

```
pomo(P,A):-johtaa(P,Y),töissä(A,Y).
```

$$\texttt{pomo} = \pi_{1,3}\sigma_{2=4}(\texttt{johtaa} \times \texttt{töissä})$$

```
CREATE VIEW pomo AS
       SELECT J.johtaja,T.työntekijä
       FROM   johtaa J,
              töissä T,
       WHERE  J.yritys=T.yritys
```

- Sama muuttuja vartalon eri predikaatinkutsuissa vastaa (luonnollista) liitosta ($\sigma_{\ldots=\ldots}(\ldots \times \ldots)$).

- Vartalossa esiintyvän muuttujan esiintyminen myös päässä vastaa projektiota ($\pi_{\ldots}$).

  Jos taas (nimetön) muuttuja esiintyy päässä muttei vartalossa, on sille vaikea keksiä tietokantatulkintaa; siksi sellaisia ei nyt sallita.

Samaa "relationaalista" ajattelua voi käyttää usein myös Prolog-ohjelmissa: vartalon kukin predikaatinkutsu hakee vastaavasta (ehkä intensionaalisesta) relaatiosta seuraavan monikon, ja koko sääntö kertoo miten nämä vartalon monikot yhdistetään pään mukaisen relaation monikoksi.

Puhtaan relationaalinen tietokanta-ajattelu ei t
aina ole mahdollista:

- Prologin suoritusjärjestys saattaa vaatia säännöille tai vartalon kutsuille jonkin tietyn järjestyksen, kun taas relaatioajattelussa järj on vapaa.

- Vastauksen kokoamisen ajatellaan tapahtuva datalogissa kokonaisia tietojoukkoja käsittelemällä yksittäisten kutsujen ratkomis sijaan.

Saman predikaatin eri säännöt vastaavat yhdist

```
käläkättäjä(X):-töissä(X,nokia).
käläkättäjä(X):-töissä(X,ericsson).
```

$$\begin{aligned}\texttt{käläkättäjä} = &\pi_1\sigma_{2=\texttt{nokia}}\texttt{töissä} \\ &\cup \pi_1\sigma_{2=\texttt{ericsson}}\texttt{töissä}\end{aligned}$$

```
CREATE VIEW käläkättäjä AS
       (SELECT T.työntekijä
        FROM   töissä T
        WHERE  T.yritys='nokia')
       UNION
       (SELECT T.työntekijä
        FROM   töissä T
        WHERE  T.yritys='ericsson')
```

Kalvoilla IV.4.4 suositeltiin, että Prolog-negaatiota

$$\texttt{not}(p)$$

sovellettaisiin vain muuttujattomiin p.

Datalogissa riittää, että kaikki negatoidussa p esiintyvät muuttujat esiintyvät samassa vartalossa myös negatoimattomina (ennen itse negaatiota).

Silloin negaatio vastaa joukkoerotusta:

```
työläinen(X):-töissä(X,Y),not(johtaa(X,Y)).
```

$$\texttt{työläinen} = \pi_1(\texttt{töissä} \setminus \texttt{johtaa})$$

```
CREATE VIEW työläinen AS
       SELECT T.työntekijä
       FROM   töissä T
       WHERE  T NOT IN johtaa
```

Näin datalog jossa on *negaatio muttei rekursiota* on "relaatiotäydellinen" kyselykieli.

(Ilman negaatiota ei saada erotusta.)

Rekursiiviset datalog-säännöt mahdollistavat se
kyselyt, joita ei voi enää kirjoittaa
perusrelaatiomallissa, kuten "ne tytäryritykset B
jotka konserni A omistaa (myös epäsuorasti)":

```
omistaa(A,B):-valvoo(A,B).
omistaa(A,B):-valvoo(A,C),omistaa(C,B).
```

Nyt haetaan relaatioiden välisen yhtälön

$$\texttt{omistaa} = \texttt{valvoo} \cup \pi_{1,4}\sigma_{2=3}(\texttt{valvoo} \times \texttt{omista}$$

ratkaisua $\texttt{omistaa} = \ldots$

Kaikista ratkaisuista halutaan *pienin* koska

- seurataan kalvojen IV.5.2 ajatusta pienimmä mahdollisesta mallista joten
- halutaan pienin mahdollinen vastausrelaatio

Osoittautuu että tämä pienin ratkaisu voidaan l
kasvattamalla aluksi tyhjää tulosta yhtälön oike
puolella kunnes se ei enää muutu:

sitten := $\emptyset$;
repeat
  ennen := sitten;
  sitten := $\texttt{valvoo} \cup \pi_{1,4}\sigma_{2=3}(\texttt{valvoo} \times \texttt{enner}$
until ennen = sitten

(Parempia mutta mutkikkaampia algoritmeja or

Tämä operaatio on itse asiassa ekstensionaalisen relaation `valvoo` *transitiivinen sulkeuma* (transitive closure, TC): $\langle a, b\rangle \in$ `omistaa` jos ja vain jos on ketju $\langle a, c_1\rangle \in$ `valvoo`, $\langle c_1, c_2\rangle \in$ `valvoo`, $\ldots, \langle c_k, b\rangle \in$ `valvoo`.

Relaatiomallia onkin ehdotettu laajennettavaksi TC-operaatiolla [DD98,§10.6].

SQL3-standardiluonnoksessa tämä laajennus on mukana rekursiivisina näkyminä [DD98,s.395] (syntaksi luennoijan oma arvaus):

```
CREATE VIEW omistaa AS
       valvoo
       UNION
       (SELECT V.konserniyritys,O.tytäryritys
        FROM   valvoo  V,
               omistaa O
        WHERE  V.tytäryritys=O.konserniyritys)
```

Negaation sovittaminen tähän rekursioon on yhä ongelma (samoin kuin kalvoilla IV.6).

Patologinen esimerkki on

```
p:-not(q).
q:-not(p).
```

jossa negaatio osallistuu rekursioon.

Kalvojen IV.5.2 mukaan $\emptyset$ ei vielä ole malli: `not` on totta mutta `p` ei.

Hieman suurempi $\{$`p`$\}$ on jo malli: `not(p)` ei ole t

Mutta myös $\{$`q`$\}$ on malli, eikä siis ole vain yhtä pienintä mallia.

Datalog-ajattelussa saadaan yhtälöpari

$$\mathtt{p} = \mathrm{full} \setminus \mathtt{q}$$
$$\mathtt{q} = \mathrm{full} \setminus \mathtt{p}$$

missä $\mathrm{full} = \{\langle\rangle\}$ on relaatiovakio: kaikki 0-paikk monikot.

Jos (yleistetty) algoritmi yrittää laskea sille pier ratkaisua (jota ei siis ole), se jää sykkimään:

kierros	p	q
0	$\emptyset$	$\emptyset$
1	full	full
2	$\emptyset$	$\emptyset$
⋮	⋮	⋮

Myös vastaava Prolog-suoritus jää ikuiseen silmukkaan.

Prolog ja datalog ovat uusien *deduktiivisten* eli (epätriviaalia) päättelyä suorittavien tietokantoj keskeisiä perusasioita.

IV.8.3 Vastausjoukot

Kalvojen IV.8 tietokanta-ajattelua tuetaan Prolog-ohjelmoinnissa kirjastopredikaateilla, joilla kaikki kyselyn vastaukset voidaan kerätä yhteen listaan ilman että ne pitäisi poimia yksi kerrallaan [B01,§7.6]:

`findall(`X`,`P`,`L`)` on intuitiivisesti "kerää listaksi L kaikki sellaiset X joilla P".

Toisin sanoen, kutsu P ratkaistaan kaikin mahdollisin tavoin, ja joka löydetyllä ratkaisulla θ (johonkin kohtaan) listaan L lisätään $X\theta$. Sama alkio voi siis toistua.

"Tulostusasu" X voi olla mikä tahansa termi joka sisältää (mm.) kutsun P muuttujia; nämä täytetään vastaavilla löydetyillä arvoilla.

`findall(esimies(P),pomo(P,A),L)` antaa listan `L` kaikista esimiehistä.

$$L \texttt{ := SELECT } X \texttt{ FROM } P$$

Jos kyselyssä P on muuttujia jotka eivät esiinny asussa X (kuten tässä A) niin niiden eri arvot (tässä alaiset) katoavat – ne projisoitiin pois.

Jos P ei ratkea on $L = $ `[]`.

`bagof(`X`,`P`,`L`)` on muuten kuten `findall/3`, mu
ne kyselyn P muuttujat Y jotka puuttuvat
asusta X käsitellään kuten Prologissa (eikä
kadoteta).

Silloin jokaiselle muuttujan Y eri arvolle a,
P ratkeaa, kootaan oma epätyhjä lista
L vastaavia tuloksia X.

$$L_a \texttt{ := SELECT } X \texttt{ FROM } P \texttt{ WHERE } Y \texttt{ = } a$$

Nykyinen a löytyy muuttujan Y arvona.
Peruutus siirtyy seuraavaan a, tai epäonnis
jos niitä ei enää ole.

Notaatiolla `bagof(`X`,`Y`^`P`,`L`)` voi määrätä muuttujan Y käyttäytymään sittenkin kirjastopredikaatin `findall/3` tapaan.

`setof(`X`,`P`,`L`)` on muuten kuin `bagof/3`, mutta
järjestetty ja sisältää kunkin X vain kerran

$$L_a \texttt{ := SELECT DISTINCT } X \texttt{ FROM } P \texttt{ WHERE } Y \texttt{ = } a \texttt{ ORDER BY } X$$

Kirjastossa on termien vertailuun predikaat
`@</2` (sekä `@=<`, `@>`, `@>=`, `==` ja `\==`).

Kirjallisuutta

AS96 H.Abelson&G.J.Sussman: *Structure and Interpretation of Computer Programs. Second Edition.* MIT Press, 1996.

AHV95 S.Abiteboul,R.Hull,V.Vianu: *Foundations of Databases.* Addison Wesley, 1995.

AHU83 A.V.Aho,J.E.Hopcroft&J.D.Ullman: *Data Structures and Algorithms.* Addison Wesley, 1983.

B01 I.Bratko: *Prolog Programming for Artificial Intelligence. Third Edition.* Addison Wesley, 2001.

CM87 W.F.Clocksin&C.S.Mellish: *Programming in Prolog. Third, Revised and Extended Edition.* Springer-Verlag, 1987.

CM98 G.Cousineau&M.Mauny: *The Functional Approach to Programming.* Cambridge University Press, 1998.

DD98 C.J.Date&H.Darwen: *Foundation for Object/Relational Databases: The Third Manifesto.* Addison Wesley, 1998.

DEC96 P.Deransart,A.Ed-Dbali&L.Cervoni: *Prolo The Standard. Reference Manual.* Springer-Verlag, 1996.

D96 R.K.Dybvig: *The Scheme Programming Language: ANSI Scheme. Second Edition.* Prentice Hall, 1996.

H92 D.Harel: *Algorithmics. Second Edition.* Ac Wesley, 1992.

H97 J.R.Hindley: *Basic Simple Type Theory.* Cambridge University Press, 1997.

KCR98 R.Kelsey, W.Clinger&J.Rees (toim.): Revi Report on the Algorithmic Language Sche *Higher-Order and Symbolic Computation* 11(1), 1998 **tai** *ACM SIGPLAN Notices* 3 1998.

L87 J.W.Lloyd: *Foundations of Logic Programming. Second, Extended Edition.* Springer-Verlag, 1987.

P02 B.C.Pierce: *Type Theory and Programming Languages.* MIT Press, 2002.

S98 S.Slade: *Object-Oriented Common Lisp*. Prentice Hall, 1998.

SS86 L.Sterling&E.Shapiro: *The Art of Prolog*. MIT Press, 1986.

S97 B.Stroustrup. *The C++ Programming Language. Third Edition.* Addison-Wesley, 1997.

T99 S.Thompson: *Haskell: The Craft of Functional Programming. Second Edition.* Addison-Wesley, 1999.