Esimerkkinä Prologin käytöstä perinteisessä ohjelmoinnissa olkoon lisäys 2-3-puuhun [AHU83,§5.4;B01,§10.1].

2-3-puu on (haku)puu, jossa

- sisäsolmuilla on 2 tai 3 alipuuta, ja
- talletettu tieto on lehdissä...
- ...jotka ovat samalla tasolla.
- Ohjelma haarautuu tapauksiin parametrihahmojen mukaan.
- Kukin tapaus on oleellisesti vastaava tasapaino(tus)ehto.
- Tietorakenneoperaatiot esitetään suoraan näillä parametrihahmoilla.
- Tietorakenteiden käsittely hahmonsovituksella on myös moderneissa funktionaalisissa kielissä kuten Haskell [T99] ja ML [CM98]. Niissä eri haarat ovat tietenkin toisensa poissulkevia, koska kullakin funktiokutsulla on vain yksi arvo.

(Listaus jatkuu yli kalvo

```
/*

* e/0: Tyhjä 2-3-puu.

* l/1: Lehti jossa alkio (=luku).

* b/3: 2-haarainen sisäsolmu, välissä
       oikeanpuoleisen haaran pienin alkio.

* t/5: 3-haarainen sisäsolmu, väleissä
       oikeanpuoleisten haarojen pienimmät alkiot.

* c/3: Kuten b/3 mutta korkeus kasvanut yhdellä.
       Algoritmi pyrkii tasoittamaan syntyneen eron.

*/

% Pääpredikaatti:
% Kun luku X lisätään puuhun T0 niin saadaan puu T.
add23(T0,X,T):-
        ins23(T0,X,T1),
        put23(T1,T).

% Jos vielä lisäyksen lopussakin on korkeus kasvanut,
% niin koko puuta on kasvatettava uudella juurella:
put23(c(T0,X,TX),
      b(T0,X,TX)).
put23(l(X),
      l(X)).
put23(b(T0,X,TX),
      b(T0,X,TX)).
put23(t(T0,X,TX,Y,TY),
      t(T0,X,TX,Y,TY)).

% Tyhjään puuhun lisäys on lehti.
ins23(e,X,l(X)).

% Lehteen lisäys on korkeutta kasvanut taimi.
ins23(l(Y),X,c(l(Y),X,l(X))):-
        Y<X.
ins23(l(Y),X,c(l(X),Y,l(Y))):-
        X<Y.

% 2-haaraiseen sisäsolmuun lisäys:
% Valitse alipuu, lisää uusi alkio siihen, ja
% pistä kasvanut alipuu takaisin paikoilleen.
ins23(b(T0,Y,TY),X,T):-
```

```
        X<Y,
        ins23(T0,X,TX),
        put2l(TX,Y,TY,T).
ins23(b(T0,Y,TY),X,T):-
        Y<X,
        ins23(TY,X,TX),
        put2r(T0,Y,TX,T).

% 3-haaraiseen sisäsolmuun lisäys:
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).

% Vaikka 2-haaraisen alipuu olisikin kasvanut korkeutta,
% ei hätää: tehdään sen tilalle 3-haarainen solmu!
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)).

put2r(T0,X,c(TX,Y,TY),
    t(T0,X,  TX,Y,TY)).
put2r(T0,X,b(TX,Y,TY),
    b(T0,X,b(TX,Y,TY))).
put2r(T0,X,t(TX,Y,TY,Z,TZ),
    b(T0,X,t(TX,Y,TY,Z,TZ))).

% Jos 3-haaraisen alipuu on kasvanut korkeutta,
% on hätä: korjaus on tehtävä lähempänä juurta!
put3l(c(T0,X,TX),Y,  TY,Z,TZ,
    c(b(T0,X,TX),Y,b(TY,Z,TZ))).
put3l(b(T0,X,TX),Y,TY,Z,ZY,
    t(b(T0,X,TX),Y,TY,Z,ZY)).
put3l(t(T0,W,TW,X,TX),Y,TY,Z,ZY,
    t(t(T0,W,TW,X,TX),Y,TY,Z,ZY)).

put3m(T0,X,c(TX, Y,  TY),Z,TZ,
  c(b(T0,X,  TX),Y,b(TY, Z,TZ))).
put3m(T0,X,b(TX,Y,TY),Z,TZ,
    t(T0,X,b(TX,Y,TY),Z,TZ)).
put3m(T0,W,t(TW,X,TX,Y,TY),Z,TZ,
    t(T0,W,t(TW,X,TX,Y,TY),Z,TZ)).

put3r(T0,X,TX, Y,c(TY,Z,TZ),
  c(b(T0,X,TX),Y,b(TY,Z,TZ))).
put3r(T0,X,TX,Y,b(TY,Z,TZ),
    t(T0,X,TX,Y,b(TY,Z,TZ))).
put3r(T0,W,TX,X,t(TX,Y,TY,Z,TZ),
    t(T0,W,TX,X,t(TX,Y,TY,Z,TZ))).

% Lisärutiineja puiden luomiseen ja katseluun:
build23([],e).
build23([X|Xs],T):-
        build23(Xs,T1),
        add23(T1,X,T).

show(T):-
        show(T,0).
show(e,_).
show(l(X),H):-
        view(X,H).
show(b(T0,X,TX),H):-
        H1 is H+5,
        view(-----,H),
        show(TX,H1),
        view(X,H),
        show(T0,H1),
        view(-----,H).
show(t(T0,X,TX,Y,TY),H):-
        H1 is H+5,
        view(-----,H),
        show(TY,H1),
        view(Y,H),
        show(TX,H1),
        view(X,H),
        show(T0,H1),
        view(-----,H).
view(A,H):-
        tab(H),
        write(A),
        nl.
```

(Listaus jatkui yli kalvo

Esimerkkinä Prologin hyödyllisyydestä etsintäongelmissa olkoon lauselogiikan kaavojen toteutumisongelma:

- Syötteenä saadaan jakamattomista atomikaavoista P_i konnektiiveilla 'ja', 'tai' ja 'ei' yhdistelty kaava.

  Esimerkiksi

  $$(\underbrace{\text{sataa}}_{P_1} \text{ tai } \underbrace{\text{paistaa}}_{P_2}) \quad \underbrace{\text{muttei}}_{\text{eli "ja ei"}} \quad \underbrace{\text{sada}}_{P_1}.$$

- Kysytään milloin se on totta.

  Esimerkiksi silloin kun ei sada mutta paistaa.

  Eli kun P_1 on valetta mutta P_2 totta.

- Riittää kirjoittaa induktiivinen *totuusmääritelmä* rekursiivisena Prolog-ohjelmana:

  "Kaava 'F ja G' on tosi jos sekä F että G on tosi", jne.

- Kun atomikaavan P_1 yhteen esiintymään laitetaan arvo `tosi` niin myös sen toinen esiintymä tulee kiinnitetyksi.

```
/*
onko(Kaava,Totuusarvo) on totta kun

1. Totuusarvo on joko 'tosi' tai 'vale'.

2. Lauselogiikan Kaava on esitetty seuraavasti:

   a. Jakamattomat atomikaavat P_i on esitetty termeinä x(V_i)
      missä V_i on oma Prolog-muuttuja kullekin i.

   b. Konjunktiot 'F/\G' eli 'F ja  G' on esitetty terminä  'ja(F,

   c. Disjunktiot 'F\/G' eli 'F tai G' on esitetty terminä 'tai(F,

   d. Negaatiot '-F' on esitetty terminä 'ei(F)'.

   Esimerkiksi kaava '(A\/B)/\-A' esitetään terminä
   'ja(tai(x(A),x(B)),ei(x(A)))'.

3. Kaava saa Totuusarvon muuttujien V_i nykyisillä sidonnoilla.

*/

onko(x(T),T).

onko(ja(F,G),tosi):-
        onko(F,tosi),
        onko(G,tosi).
onko(ja(F,G),vale):-
        onko(tai(ei(F),ei(G)),tosi).

onko(tai(F,_),tosi):-
        onko(F,tosi).
onko(tai(_,G),tosi):-
        onko(G,tosi).
onko(tai(F,G),vale):-
        onko(ja(ei(F),ei(G)),tosi).

onko(ei(F),tosi):-
        onko(F,vale).
onko(ei(F),vale):-
        onko(F,tosi).
```

IV.4 Kirjastopalveluita

Tutustutaan Prolog-standardin peruspalveluihin listankäsittelyyn, aritmetiikkaan, samastukseen, negaatioon sekä syöttöön ja tulostukseen [B01,§2.7,3.1-3.2,3.4,5,6,7.2-7.3]. Toteutukset tarjoavat usein enemmänkin, varsinkin listoille.

IV.4.1 Listat

Myös Prolog tarjoaa kalvojen II.8 kaltaiset listat.

Prolog kirjoittaa listansa *alkiot pilkuilla eroteltuina hakasulkeisiin*: `[tämä,on,listavakio]`.

Tyhjä lista on siis `[]`.

Erillistä lainausmerkkimekanismia ei nyt tarvita, koska listan erottaa muista termeistä hakasulkeiden perusteella.

Listoja voi käsitellä kalvojen IV.5.1 *listahahmoil*

$$[A_1,\ldots,A_k|D]$$

on hahmo sellaiselle listalle, jossa on ainakin k alkiota.

Jos loppuhahmo '$|D$' puuttuu, on kyseessä hah tasan k-alkioiselle listalle (eli loppuhahmoksi ote '`|[]`').

Hahmossa kukin A_i ja D ovat Prolog-termejä.

Listan i. alkiota yritetään samastaa termin A_i kanssa, ja loppulistaa termin D kanssa:

```
?- [tämä,on,lista]=[A1,A2|D].

A1 = tämä
A2 = on
D = [lista]
```

Näillä hahmoilla voidaan purkaa listoja kuten Scheme-kirjastofunktioilla `cd`$^{k-1}$`ar` ja `cd`^{k}`r`.

Niillä voidaan myös *rakentaa* listoja lukemalla n toisin päin: muodosta uusi lista lisäämällä listan D alkuun alkiot $A_k,\ldots,A_1$.

Esimerkiksi kahden listan liittäminen kolmanneksi on:

```
conc([],Y,Y).
conc([X|Xs],Y,[X|Z]):-
        conc(Xs,Y,Z).
```

- Jos ensimmäinen lista on tyhjä, niin toinen ja kolmas lista ovat samat.

- Jos ensimmäisellä listalla on ensimmäinen alkio `X`, niin kolmas lista alkaa tällä alkiolla ja jatkuu listana `Z`, joka puolestaan on ensimmäisen loppulistan `Xs` ja koko toisen listan `Y` liitos.

Tuloksella voi paitsi liittää myös *jakaa* listoja:

```
?- conc([1],[2],Z).

Z = [1, 2] ;

No
?- conc(X,Y,[1,2]).

X = []
Y = [1, 2] ;

X = [1]
Y = [2] ;

X = [1, 2]
Y = [] ;

No
?-
```

(Eikä ihme, sehän on kalvojen IV.2 `p/3` listoille. . .)

IV.4.2 Aritmetiikka

Kalvoilla I.3 nähtiin Prolog-ohjelma (yksinkertai derivointiin.

Sen sääntöjen päissä käytettiin aritmeettisia operaattoreita `+`, `*`, `/` ja `**` (potenssiin korotus) *symbolisina* kaksipaikkaisina funktoreina (jotka kirjoitettiin argumenttiensa väliin matematiikan tavalliseen tapaan).

Niiden avulla ohjelma haarautui ensimmäisen argumentin hahmon perusteella oikeaan sääntöö tuotti toiseen argumenttiin vastaavan derivaattatermin vartalosta saatujen dervivaattatermien avulla.

Mutta ohjelman toisen säännön vartalossa eksponenttia `N` käytettiinkin *numerona*, josta vähennettiin `1`, jotta tuloksena saataisiin uusi eksponentti `N1`.

Tämän sai aikaan (myös väliin kirjoitettu) kirjastopredikaatti V `is` E joka on totta kun (i) on nykyisellä sidonnalla sellainen aritmeettinen lauseke jonka arvo voidaan laskea ja (ii) V sopii tuohon laskettuun arvoon.

Jos (i) ei pädekään, seuraa ajonaikainen virhe.

Kalvojen I.3 ohjelmassa kutsu `N1 is N-1` on siis:

1. Tarkista, että lauseke `N-1` on nykyisessä sidonnassa aritmeettinen:

   - Ei vapaita muuttujia.
   - Vain aritmeettisia funktoreita (kuten +/2, */2, //2, **/2,...).
   - Vain numerovakioita.

   Silloin lausekkeella on numeerinen arvo a.

   Muuten kyseessä on ajonaikainen virhe.

2. Yritä sitoa `N1` arvoon a.

Prolog-kielessä on äärellisen tarkkuuden kokonais- ja liukuluvut. Liian suurista kokonaisluvuista tulee automaattisesti liukulukuja, liian suurista liukuluvuista ääretön atomi `$Infinity`.

Myös (jälleen väliin kirjoitettavat) aritmeettiset vertailut =:=/2 ($=$), =\=/2 ($\neq$), >/2 ($>$), >=/2 ($\geq$), </2 ($<$) ja =</2 ($\leq$) laskevat argumenttiensa numeroarvot samalla tavalla.

Maali `?-1<X,X<3.` antaa siis virheen.
(Logiikkaohjelmointi voidaan tosin laajentaa tällaisiin *reaalilukurajoitteisiin* [B01,§14.2].)

IV.4.3 Samastus

Kalvoilla III.2 esiteltiin keskeinen samastusopera `unify`, jolla sääntöjärjestelmä teki annetuista termeistä samoja sitomalla niiden muuttujia.

Samastusalgoritmin haara `uvc` eli "vapaa muutt x vastaan rakenteinen termi t" käytti testiä `binding:occurs?` varmistamaan, ettei x esiinny termissä t. Muutenhan sidonnan $x = t$ lisäämine aiheuttaisi kehän.

Tämä *esiintymistarkistus* (occurs check)

- hidastaa samastusta selvästi
- on harvoin tarpeen käytännön ohjelmoinniss

joten Prologissa se *jätetään pois – vaikka silloir kalvojen IV.5 puhdas rakenne likaantuu!*

Puhtaan samastuksen voi tehdä kirjastopredikaa `unify_with_occurs_check`/2.

Myös Prologin omaa samastusta voi kutsua nimellä:

- =/2 luetaan "samastuvat" (eli "ovat samat"), jota käytettiin jo kalvojen IV.4.1 esimerkissä. (On myös ==/2 eli "taysin samat ilman samastustakin".)

- \=/2 luetaan "eivät samastu". (On myös \==/2.)

Esiintymistarkistuksen puuttuminen tekee näistä hieman arvaamattomia: ne saattavat esimerkiksi päätyä ikuiseen silmukkaan.

Prolog tarjoaa myös (jo kalvoilla IV.3 nähdyn) *nimettömän muuttujan* `_` (eli pelkän alaviivan) joka samastuu aina mutta arvoa ei muistella. Esimerkiksi `f(a,_,_)` samastuu jokaiseen `f(a,`t_1`,`t_2`)` vaikka t_1 ja t_2 eivät samastuisikaan keskenään.

Samastuksen kuluessa rakenteisia termejä puretaan osiinsa. Sama palvelu tarjotaan (väliin kirjoitettavalla) kirjastopredikaatilla `=../2` (nimeltään "`univ`"). Se on seuraava syntaktinen relaatio:

$$f(t_1,\ldots,t_k) \texttt{ =.. } [f,t_1,\ldots,t_k]$$

(Myös $k = 0$ käy.) Sitä voidaan käyttää myös "oikealta vasemmalle" eli kokoamaan termi annetusta listasta.

IV.4.4 Negaatiosta

Kalvojen IV.3 sukupuuesimerkissä olisimme halunneet kirjoittaa säännöt "mies on terve (eli jos hän *ei* ole sairas (eli `ill`)". Kuitenkin kirjoit nämä faktat erikseen.

Syynä on se, että *Prolog-kielen negaatio (eli "ei...") on jotakin muuta kuin odottaisi.* Tähä palataan myöhemmin tutustuttuamme kielen merkitysoppiin tarkemmin.

Prolog-kielessä on negaatio `not(`p`)` missä p on kiellettävä alimaali. Sukupuuesimerkin sääntö ol siis `well(X) :- not(ill(X)).`

(Voi kirjoittaa myös ilman sulkuja `not ill(X).`)

Tämä `not/1` on eräänlainen korkeamman kertalu predikaatti, koska se ottaa argumentikseen toise suoritettavan eli todistettavan faktan p. Prolog kuitenkin "filosofialtaan" ensimmäisen kertaluvu kieli, jossa on vain termejä ja niiden predikaatte Omien korkeamman kertaluvun laajennusten ohjelmointia olisi siis syytä välttää "eleganssisyi

Prolog-negaatio `not(`p`)` on *negaatio epäonnistumisena (Negation as Failure):* yritetään osoittaa p, ja jos tämä ei onnistu, niin silloin katsotaan, että on osoitettu ei-p.

Negaatio epäonnistumisena *toimii kun p ei sisällä vapaita muuttujia sillä hetkellä kun kutsu* `not(`p`)` *tehdään.*

Muuten tulos saattaa olla intuitiomme vastainen, joten tästä negaationkäyttöperiaatteesta kannattaa pitää kiinni! Siis: *tuota ensin muuttujan X lopullinen arvo* ja vasta sitten testaa sitä arvoa negatiivisella ehdolla `not(q(`X`))`!

Monissa Prolog-toteutuksissa negaation nimi onkin \+ korkostamaan sitä, ettei kyseesä olekaan "oikea" (eli kielellisen intuitiomme mukainen) negaatio.

IV.4.5 Haun ohjailusta

Kalvojen IV.4.4 negaatio epäonnistumisena voit ohjelmoida predikaattina

```
not(P) :- call(P),!,fail.
not(_).
```

- `call(P)` tulkitsee muuttujan `P` arvon faktana tekee vastaavan aliohjelmakutsun.

  Joissakin toteutuksissa riittääkin kirjoittaa suoraan `P`.

  Vertaa Scheme-kirjastofunktio `eval` kalvoilla II.9.2.

- Huutomerkki '`!`' eli *katkaisupredikaatti* (Cut ohjaa Prolog-ohjelman suoritusta eli todistu hakua.

  Kalvojen IV.3 aliohjelmatulkinnassa se voida lukea "jos suoritus peruuttaa **taaksepäin** yl aliohjelmassa nimeltä q esiintyvän huutomer niin suoritus putoaa suoraan ulos aliohjelma alareunasta yrittämättä sen muita vaihtoeht

- Nyt huutomerkki sanoo siis "`P` onnistui yhdellä tavalla, älä turhaan etsi muita tapoja, tämä tapa kelpaa minulle".

  Huutomerkillä voi siis *parantaa tehokkuutta* kertomalla järjestelmälle, että nyt on löydetty se tulos, joka haluttiin — etsintä voidaan tältä osin lopettaa. Tähän palataan myöhemmin.

- `fail` epäonnistuu aina (eli sillä ei ole määritelmää).

  Negaation määritelmän lisäksi `fail` on hyödyllinen esimerkiksi silloin, kun Prolog-ohjelman täytyy tehdä sivuvaikutuksenaan jotakin. Tähän palataan myöhemmin tietokantojen yhteydessä.

Siis `not`-predikaattimme luetaan seuraavasti:

"Jos aliohjelmakutsu `P` onnistuu, niin rajoita `not(P)` siihen ratkaisuun, jota ei ole.

Muuten `not(P)` onnistuu."

Siispä se on "negation as `failure`".

IV.4.6 Syöte ja tulostus

Prolog-kielen syöte ja tulostus noudattaa samaa ajattelutapaa kuin Scheme-kielessäkin (kalvot II.9.7–II.9.8):

- Syöte ja tulostus voidaan ohjata portteihin kutsutaan *virroiksi* (streams)]

- Voidaan lukea (`read`/1 yms.) ja kirjoittaa (`write`/1 yms.) kokonaisia Prolog-termejä.

Yksityiskohdat sivuutetaan. (Toteutukset eivät tue kaikkia standardin piirteitä.)

Periaatteessa Prologin pitäisi pystyä *peruuttam* myös syötteenluku- ja tuloksenkirjoitusoperaatio mutta tämä olisi käytännössä liian vaikeaa: luet syöte pitäisi palauttaa takaisin ja kirjoitettu tul pyyhkiä pois.

Näin *ei* siis tehdä, joten syötteen ja tulostuksen käyttö vaatii Prolog-suoritusjärjestyksen tuntem

Esimerkiksi kutsu `read(`X`)` lukee muuttujaan X (nykyisestä syöteportista) seuraavan Prolog-termin t ja sen perässä olevan pisteen '.'.

Peruutettaessa ei löydy muita ratkaisuja ja t (pisteineen) pysyy luettuna.

Jos X onkin termi u, niin t luetaan (pisteineen) ensin, ja sitten yritetään samastaa t ja u. Jos samastus onnistuu, niin kutsukin onnistuu; muuten kutsu epäonnistuu – mutta molemmissa tapauksissa t on luettu (pisteineen)!

Lukea (ja kirjoittaa) voi myös sääntöjä: ne tulkitaan termeiksi `':-'(pää,vartalo)` missä

- funktori `:-/2` kirjoitetaan väliin
- vartalo on joko yksinäinen termi tai sitten `termi,vartalo`

ja muuttujat numeroidaan uudelleen.

Prolog-ohjelman lataus(kirjasto)predikaatti `consult('tiedostonimi')` (vertaa Scheme-kalvoja II.9.9) lukeekin `read`illa (vertaa Scheme-kalvoja II.9.8) termejä, ja liittää ne nykyisen ohjelman loppuun.

IV.5 Semantiikka

Logiikkaohjelmoinnissa ohjelmien merkitystä voi kuvailla kolmesta eri näkökulmasta:

Suorituslähtöisesti muiden ohjelmointikielten tapaan [B01,§2.4].

"Mitä askeleita ohjelma tekee?"

Kalvojen IV.2 ohjelma **sum** siirtää toisen argumenttinsa kolmanteen, ja sen päälle kai ensimmäisen `s`-funktorit.

Näin tehtiin kalvoilla IV.3. Kalvojen III tulkk tämän näkökulman toteutus.

Malliteoreettisesti käyttäen *loogisen totuuden* käsitettä [B01,§2.3;L87,§6].

"Millaisen maailman ohjelma kuvailee?"

Ohjelma **sum** kertoo mitä merkitsee väite "kolmas luku on kahden ensimmäisen summ

(Loogiset yksityiskohdat sivuutetaan; logiikk opiskelleet kohtaavat tuttujen käsitteiden erikoistapauksia.)

Todistusteoreettisesti käyttäen loogisen *todistuksen* käsitettä [L87,§7-10].

Ohjelma **sum** on aksiomatisointi jolla voidaan todistaa näitä väitteitä "kolmas luku on kahden ensimmäisen summa".

- Malliteoria kuvaa ohjelman merkityksen ilman viittauksia mihinkään laskentasääntöihin.

  (Ohjelma on siis symbolikasauma, jolle ihmisellä on mielessään tulkinta...)

- Todistusteoria kuvaa sellaiset mahdolliset laskentasäännöt, jotka ovat malliteorian mukaisia.

  (...joten säännöt ovat siis niitä symbolisia operaatioita, jotka eivät sodi tulkintaa vastaan...)

- Suorituskone toteuttaa (jonkin) strategian jolla todistusteorian sallimia laskentasääntöjä yritetään soveltaa laskentatehtäviä ratkaistaessa.

  (...joten ohjelmoidaan tietokone ratkomaan laskutehtäviä näillä operaatioilla!)

Samaa kolminaisuutta käytetään yleisemminkin ohjelmointikielten teoriassa, mutta logiikkaohjelmoinnissa se näkyy harvinaisen puhtaana.

IV.5.1 Suoritussemantiikka

Tarkastellaan predikaatin p/k määritelmää n säännöllä:

$$p(t_{(1,1)},\ldots,t_{(1,k)}) \text{ :- } q_{(1,1)},\ldots,q_{(1,m_1)}.$$
$$\vdots$$
$$p(t_{(n,1)},\ldots,t_{(n,k)}) \text{ :- } q_{(n,1)},\ldots,q_{(n,m_n)}.$$

Sen voi lukea määritelmänä aliohjelmalle nimelt
jolla on k parametria.

Jokaisessa haarassa $1 \leq i \leq n$ termit $t_{(i,1)},\ldots,t_{(}$
määräävät *hahmon* johon aliohjelmakutsun $p(\tau_1,\ldots,\tau_k)$ argumenttien $\tau_1,\ldots,\tau_k$ on *sovittava*
(kalvojen III.2) samastuksen mielessä.

Haara i on *mahdollinen* jos argumentit sopivat hahmoon; muuten haara sivuutetaan. Hahmo si
kertoo minkä "muotoisille" syötteille τ_j tämä ha
on tarkoitettu.

Koska nyt samastetaan argumentteja parametrihahmoihin, on mukana myös sidonta θ
laajennetaan suorituksen edetessä, ja joka hoita
parametrinvälityksen. (Vrt. kalvojen II.9.1 ympäristöt.)

Loogikot kirjoittavat $\tau\theta$ operaatiomme `(binding:expand` τ θ`)` vastineen.

(Ali)ohjelman lukuperiaate on tuttu "ylhäältä alas, vasemmalta oikealle":

Etsitään ensimmäinen mahdollinen haara i, eli sellainen jolla operaation

$$\theta_0 = (\texttt{unify}\ p(\tau_1, \ldots, \tau_k)\ p(t_{(i,1)}, \ldots, t_{(i,k)})\ \theta)$$

vastine onnistuu.

Ensin kuitenkin nimettiin haaran i muuttujat uudelleen, jotta vanhat muuttujanarvot θ ja uudet parametrit eivät sekoittuisi keskenään.

Jatketaan kokeiltavan haaran i vartaloon, eli

1. tehdään samalla tavalla kutsu $q_{(i,1)}$ sidonnalla θ_0, josta saadaan vastaukseksi sidonta θ_1,

2. tehdään kutsu $q_{(i,2)}$ sidonnalla θ_1, josta saadaan sidonta θ_2,...

Lopulta vastataan kysyjälle että kutsu $p(\tau_1, \ldots, \tau_k)$ laajensi alkusidonnat θ loppusidonnoiksi θ_{m_i}.

Entä jos jokin seuraava kutsu $q_{(i,j+1)}$ ilmoittaakin, ettei se pysty laajentamaan edelliseltä kutsulta $q_{(i,j)}$ saamaansa sidontaa θ_j sidonnaksi θ_{j+1}?

Silloin *peruutetaan* (backtrack) takaisin edelliseen kutsuun $q_{(i,j)}$, ja pyydetään siltä *jotakin toista* sidonnan θ_{j-1} laajennusta sidonnaksi θ'_j.

Jos tämän kokeiltavan haaran i vartalosta peruutetaan takaisin päähän asti, niin tämä kut $p(\tau_1, \ldots, \tau_k)$ sidonnoilla θ on sittenkin suoritetta käyttäen jotakin toista haaraa.

Jatketaan siis seuraavan mahdollisen haaran etsimistä haarasta $i+1$ alkaen. Jos sellainen lö niin kokeillaan sitä.

Jos taas haarat loppuvat kesken, niin vastataan kysyjälle "kutsu $p(\tau_1, \ldots, \tau_k)$ ei pysty laajentama sidontaa θ". Tämä aiheuttaa peruutusta vuoros kysyjässä.

Vastaavasti myös kysyjältä voi tulla pyyntö "va kutsu $p(\tau_1, \ldots, \tau_k)$ laajensikin sidontansa θ onnistuneesti sidonnaksi θ_{m_i} haaralla i, ei tulos sopinutkaan myöhempään laskentaan, joten voi sittenkin saada *jonkin toisen* vaihtoehdon θ'_{m_i}?"

Silloin peruutetaan takaisin viimeiseen onnistuneeseen kutsuun $q_{(i,m_i)}$, ja yritetään laaj sen saamaa sidontaa θ_{m_i-1}.

Perimmäisenä kysyjänä on käyttäjä, joka saatta tyytymätön hänelle tulostettuun vastaukseen.

Prolog-ohjelmien suorituksessa vuorottelee siis:

Etenemisvaihde jonka aikana suoritetaan aina heti seuraava mahdollinen haara.

Kun näin ajaudutaan umpikujaan, kytketään päälle...

Peruutusvaihde jonka aikana ohjelman suoritusta "kelataan takaisinpäin" edelliseen sellaiseen kohtaan, joka olisi voitu tehdä toisinkin.

Siinä kohdassa valitaankin nyt toisin, ja jatketaan taas etenemisvaihteella...

Samoin aliohjelmilla ei ole kiinteitä syöte- ja tulosparametreja, vaan jokainen kutsu lukee syötteenään sille annettua sidontaa, ja tuottaa tuloksenaan siitä laajennettua versiota.

Esimerkiksi predikaattia `p/3` voi käyttää laskemaan yhteen $1+2$ kutsulla `p(s(o),s(s(o)),Z)` eikä mitään peruutusta tapahdu, koska etenemisvaihtoehtoja on aina tasan yksi, ja lopuksi onnistutaan saamaan vastaus `Z=s(s(s(o)))`.

Kääntäen luku 3 voidaan hajoittaa summiksi $0+3, 1+2, 2+1, 3+0$ kutsulla `p(X,Y,s(s(s(o))))` jossa kukin käyttäjän pyytämä peruutus tuottaa seuraavan vastauksen.

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

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

?- consult('sum.pl').
% sum.pl compiled 0.01 sec, 652 bytes

Yes
?- p(s(o),s(s(o)),Z).

Z = s(s(s(o))) ;

No
?- p(X,Y,s(s(s(o)))).

X = o
Y = s(s(s(o))) ;

X = s(o)
Y = s(s(o)) ;

X = s(s(o))
Y = s(o) ;

X = s(s(s(o)))
Y = o ;

No
?-
```

IV.5.2 Mallisemantiikka

Olkoon L tarkasteltava ohjelma. Sen sisältämät funktorit voidaan (yleensä) jakaa predikaatteihin P_L ja muihin funktoreihin F_L.

Esimerkiksi $P_{\mathbf{sum}} = \{\mathtt{p}/3\}$ ja $F_{\mathbf{sum}} = \{\mathtt{o}/0, \mathtt{s}/1\}$.

(Ranskalainen Jacques Herbrand kehitti todistusteoriaa 1930-luvulla.)

Annetun ohjelman L Herbrandin...

Universumi U_L on ne muuttujattomat termit jotka voidaan muodostaa ilman predikaatteja, eli pienin sellainen joukko, että

- jos $a/0 \in F_L$, niin myös $a/0 \in U_L$ ja
- jos $f/k \in F_L$ ja $\tau_1, \ldots, \tau_k \in U_L$ (missä $k > 0$), niin myös $f(\tau_1, \ldots, \tau_k) \in U_L$.

$$\begin{aligned} U_{\mathbf{sum}} &= \{\mathtt{o}\} \\ &\cup \{\mathtt{s(o)}\} \\ &\cup \{\mathtt{s(o)}, \mathtt{s(s(o))}\} \\ &\vdots \\ &= \{\mathtt{s}^n\mathtt{o} : n \in \mathbb{N}\}\,. \end{aligned}$$

Kanta B_L on ne faktat jotka voidaan muodosta universumin U_L termeistä, eli

$$B_L = \{p(\tau_1, \ldots, \tau_k) : p/k \in P_L\}\,.$$
$$B_{\mathbf{sum}} = \left\{\mathtt{p(s}^a\mathtt{o,s}^b\mathtt{o,s}^c\mathtt{o)} : a, b, c \in \mathbb{N}\right\}.$$

Malli on mikä tahansa kannan B_L osajoukko, jo on poimittu tosiksi ainakin niin paljon fakto että ohjelman L kaikki *lauseet* (clauses, eli säännöt ja faktat) ovat tosia:

Olkoon θ mielivaltainen muuttujien sidonta universumin U_L alkioihin.

- Malliin poimitaan (muuttujaton) fakta $p\theta$ ohjelman L jokaiselle faktalle p.

  Haluamme näet pitää totena faktaa p *ka mahdollisilla sen muuttujien saamilla arv*

- Jos malliin on jo poimittu faktat $q_1\theta, \ldots,$ ohjelmassa on sääntö p**:-**q_1**,...,**q_n**.** niin poimitaan myös $p\theta$.

  Haluamme näet pitää totena *ehtolausett*

  jos q_1 **ja ...ja** q_n **niin myös** p

  jonka loogikko kirjoittaisi *implikaationa*

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

  missä $\vec{X}$ kattaa kaikki säännön muuttuja (Ns. *Hornin lauseena*.)

Osoittautuu että on olemassa *pienin* malli $M_L \subseteq B_L$, ja se saadaan poimimalla mukaan vain ne faktat, jotka nämä ehdot pakottavat, eikä mitään muuta.

Tämä M_L on ohjelman L deklaratiivinen merkitys, ja ohjelman L suorittaminen annetulla (muuttujattomalla) maalifaktalla p on kysymykseen "Onko $p \in M_L$?" vastaamista.

(Tarkkaan ottaen joukkoon F_L on lisättävä myös maalifaktan p funktorit...)

Kalvojen IV.5.1 menetelmä on yksi algoritmi vastauksen laskemiseksi, kuitenkin *epätäydellinen*:

- algoritmi saattaa joutua ikuiseen silmukkaan löytämättä malliteorian mukaista vastausta
- koska algoritmi kokeilee sääntöjä kiinteässä järjestyksessä, joka saattaa olla tällä kertaa väärä. Malliteoriassa säännöt ovat tasa-arvoisia, mutta sellainen algoritmi söisi liikaa työmuistia.

Faktasta: $\mathtt{p(o,s^b o,s^b o)} \in M_{\mathbf{sum}}$ kaikilla $b \in \mathbb{N}$.

Säännöstä: jos $\mathtt{p(s^a o,s^b o,s^c o)} \in M_{\mathbf{sum}}$ niin myös $\mathtt{p(s^{a+1} o,s^b o,s^{c+1} o)} \in M_{\mathbf{sum}}$.

Minimaalisuudesta: muuta ei mallissa $M_{\mathbf{sum}}$ ole.

Osoitetaan $M_{\mathbf{sum}} = \left\{\mathtt{p(s^a o,s^b o,s^{a+b} o)} : a, b \in \mathbb{N}\right\}$

"$\supseteq$" : Olkoon $\mathtt{p(s^a o,s^b o,s^{a+b} o)}$ mielivaltainen. Näytetään induktiolla että $\in M_{\mathbf{sum}}$:

$a = 0$: Seuraa suoraan faktasta.

$a > 0$: Seuraa induktio-oletuksesta

$$\mathtt{p(s^{a-1} o,s^b o,s^{a-1+b} o)} \in M_{\mathbf{sum}},$$

ja siihen sovelletusta säännöstä.

"$\subseteq$" : Olkoon $\rho = \mathtt{p(s^a o,s^b o,s^c o)} \in M_{\mathbf{sum}}$ mielivaltainen. Näytetään induktiolla suuree suhteen, että $c = a + b$.

Miten ρ ilmestyi joukkoon $M_{\mathbf{sum}}$?

- Jos käytettiin faktaa, niin $a = 0$ ja $b = c$, siis $c = a + b$.
- Jos käytettiin sääntöä, niin sen vartalolle

$$\mathtt{p(s^{a-1} o,s^b o,s^{c-1} o)} \in M_{\mathbf{sum}},$$

ja siitä induktio-oletuksella $c - 1 = a - 1$ - $c = a + b$.

Minimaalisuuden nojalla *muita tapoja ei ole*

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.