Logică Matematică şi Computaţională
Suport teoretic pentru laborator

Claudia MUREȘAN
cmuresan@fmi.unibuc.ro, claudia.muresan@g.unibuc.ro

Universitatea din București
Facultatea de Matematică și Informatică
București

2024–2025, Semestrul II

1. Introducere
2. Inițiere în programarea în Prolog
3. Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului
4. Să exemplificăm noțiunile anterioare într–un program în Prolog
5. Algoritmul de unificare
6. Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?
7. Mnemonic din sintaxa calculului cu predicate clasic
8. Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor
9. Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog
10. Rezoluția în Prolog

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Prescurtări uzuale care vor fi folosite în lecții

- **i. e.** (*id est*) = adică
- **a. î.** = astfel încât
- **ddacă** = dacă și numai dacă
- **ș. a. m. d.** = și așa mai departe
- $\underline{\quad //\quad}$: să se demonstreze că
- : contradicție

Paradigme ale programării calculatoarelor

Programarea imperativă:

- programatorul trebuie să–i spună calculatorului, pas cu pas, ce să facă: "parcurge cu un indice această mulțime, la fiecare iterație verifică această condiție..." etc..

Limbaje de programare imperativă: *Pascal, C, Java* etc..

Programarea logică/declarativă:

- programatorul descrie un cadru de lucru, și cere calculatorului să rezolve o cerință în acel cadru;
- pe baza unui backtracking și a altor tehnici de programare încorporate în interpretorul/compilatorul limbajului de programare logică folosit, calculatorul determină proprietățile pe care le poate folosi pentru a rezolva cerința respectivă și ordinea în care trebuie să le aplice.

Limbaje de programare logică/declarativă

- *Prolog* (PROGRAMMING IN LOGIC): bazat pe **predicate**, pe relații între obiecte; utilizat pentru jocuri/deducții logice, în procesarea limbajului natural etc.;
- *CafeObj, Maude:* bazate pe **specificații**, pe descrierea unor tipuri de structuri algebrice; destinate demonstrării automate de proprietăți matematice; utilizate și în verificarea sistemelor software;
- *Haskell:* bazat pe **funcții** și **recursii**;

recursia este foarte importantă în toate limbajele de programare logică: este principalul mijloc de calcul, înlocuind, de exemplu, instrucțiunile repetitive

- etc..

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Siteul SWI–Prolog

https://www.swi-prolog.org

Din josul paginii de la această adresă:

- se poate descărca SWI–Prolog;
- se poate căuta în documentația SWI–Prolog;
- se poate accesa versiunea online a SWI–Prolog.

De exemplu, dând o căutare după "boolean expressions", găsim:

SWI-Prolog -- Manual

swi-prolog.org/pldoc/man?section=clpb-exprs

Documentation
Reference manual
The SWI-Prolog library
library(clpb): CLP(B): Constraint L
Introduction
Boolean expressions
Interface predicates
Examples
Obtaining BDDs
Enabling monotonic CLP(B)
Example: Pigeons
Example: Boolean circuit
Acknowledgments
CLP(B) predicate index
Packages

A.8.2 Boolean expressions

A *Boolean expression* is one of:

`0`	false
`1`	true
variable	unknown truth value
atom	universally quantified variable
~ *Expr*	logical NOT
Expr + *Expr*	logical OR
Expr * *Expr*	logical AND
Expr # *Expr*	exclusive OR
Var ^ *Expr*	existential quantification
Expr =:= *Expr*	equality
Expr =\= *Expr*	disequality (same as #)
Expr =< *Expr*	less or equal (implication)
Expr >= *Expr*	greater or equal
Expr < *Expr*	less than
Expr > *Expr*	greater than
`card(Is,Exprs)`	cardinality constraint (*see below*)
`+(Exprs)`	n-fold disjunction (*see below*)
`*(Exprs)`	n-fold conjunction (*see below*)

where *Expr* again denotes a Boolean expression.

The Boolean expression `card(Is,Exprs)` is true iff the number of true expressions in the list *Exprs* is a member of the list *Is* of integers and integer ranges of the form `From-To`. For example, to state that precisely two of the three variables *X*, *Y* and *Z* are `true`, you can use `sat(card([2],[X,Y,Z]))`.

`+(Exprs)` and `*(Exprs)` denote, respectively, the disjunction and conjunction of all elements in the list *Exprs* of Boolean expressions.

Atoms denote parametric values that are universally quantified. All universal quantifiers appear implicitly in front of the entire expression. In residual goals, universally quantified variables always appear on the right-hand side of equations. Therefore, they can be used to express functional dependencies on input variables.

ENG 6:08 PM 2/12/2020

Structura unui program în Prolog

Un program în Prolog este format din **clauze**; acestea sunt de trei tipuri:

1. **fapte**:

propoziție.
predicat(*listă de argumente: variabile, constante sau termeni compuși*).

predicatele exprimă relații între obiecte, proprietăți ale obiectelor, și au valori booleene; termenii compuși sunt formați din operatori (nu neapărat booleeni) aplicați unui număr nenul de argumente;
propozițiile sunt predicatele fără variabile, i.e. constantele booleene;
faptele semnifică proprietăți întotdeauna adevărate;

2. **reguli**:

fapt :- succesiune de fapte.

faptele din partea dreaptă a unei reguli pot fi separate prin virgulă (care reprezintă *conjuncția logică*), punct și virgulă (*disjuncția logică*, având prioritate mai mică decât conjuncția) sau alți conectori logici (a se vedea slideul anterior); simbolul ":-" (două puncte și minus) este numit "neck";

(semnificația unei reguli)

are loc acest fapt :- dacă au loc aceste fapte.

3. **întrebări (interogări)**: formate din **scopuri**, constând în predicate care trebuie satisfăcute de către Prolog: prin scopurile din interogări cerem

Prologului să determine dacă un predicat e adevărat sau valori ale variabilelor pentru care acel predicat e adevărat.

Faptele și **regulile** formează **baza de cunoștințe**.

Numele de **variabile** în Prolog încep cu *literă mare* sau *underscore* (_).
Variabilă nedenumită: *underscore* (_):

- simbol generic pentru variabile care apar o singură dată într–un fapt sau o regulă;
- sunt folosite doar pentru a indica locații de variabile, nu și pentru a lucra cu acele variabile.

Orice alt nume, inclusiv șiruri de caractere cuprinse între apostrofuri, denumește o constantă, o funcție cu argumente (i.e. operație cu cel puțin un operand) sau un predicat.

Sintaxa pentru liste în Prolog:

$[\,]$	lista vidă
$[elem_1, elem_2, \ldots, elem_n]$	lista formată din elementele $elem_1, elem_2, \ldots, elem_n$
$[elem_1, elem_2, \ldots, elem_n \mid T]$	lista cu primele n elemente $elem_1, elem_2, \ldots, elem_n$ și coada T

[ ] este o constantă predefinită în Prolog.
Listele nevide se construiesc cu operatorul binar predefinit în Prolog [|], având ca argument stâng capul listei (un singur element), iar ca argument drept coada listei (o listă).

Exemplu

$$[1,2,3,4] = [1,2,3,4|[\,]] = [1|[2,3,4]] = [1,2|[3,4]] = [1|[2,3|[4]]] = [1|[2,3|[4|[\,]]]]$$

Toate acestea sunt scrieri alternative permise de Prolog pentru următorul termen Prolog: $[1|[2|[3|[4|[\,]]]]]$, având acest arbore asociat:

Am folosit definiția uzuală pentru arborele asociat unei expresii:

(Arborii asociați termenilor (vom vedea): definiți recursiv)

arborele asociat unei variabile V:

frunză etichetată cu V:

arborele asociat unei constante ct:

frunză etichetată cu ct:

arborele asociat unui termen compus $f(arg_1, arg_2, \ldots, arg_n)$, unde: $n \in \mathbb{N}^*$, f: operator n–ar, $arg_1, \ldots, arg_n$: termeni

Exemplu

Cum putem scrie predicate pentru:

- calculul lungimii unei liste;
- concatenarea a două liste?

Să scriem un predicat *lung*(L, N), care este satisfăcut ddacă:

- L este o listă, N este un număr natural și
- N este lungimea listei L, adică numărul elementelor lui L:

$$\mathrm{lung}([\,], 0).$$
$$\mathrm{lung}([_|\mathrm{T}], \mathrm{N}) \ :- \ \mathrm{lung}(\mathrm{T}, \mathrm{K}), \ \mathrm{N} \text{ is } \mathrm{K} + 1.$$

Operatorul **is** produce executarea calculului în acea expresie aritmetică.
Înlocuiți **is** cu **=** și vedeți ce obțineți!
Și un predicat *concat*$(L1, L2, L)$, care este satisfăcut ddacă:

- $L1$, $L2$ și L sunt liste și
- concatenarea listei $L1$ cu $L2$ este L:

$$\mathrm{concat}([\,], \mathrm{L}, \mathrm{L}).$$
$$\mathrm{concat}([\mathrm{H}|\mathrm{T}], \mathrm{L}, [\mathrm{H}|\mathrm{M}]) \ :- \ \mathrm{concat}(\mathrm{T}, \mathrm{L}, \mathrm{M}).$$

Încărcăm acest program în versiunea online a SWI–Prolog

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help

359 users online

Search

Program

```
% Aici: fapte si reguli.

lung([],0).                          % fapt: intotdeauna adevarat
lung([_|T],N) :- lung(T,K), N is K+1.   % regula: are structura:
                        % are loc acest fapt :- daca au loc aceste fapte.

/* Daca denumim capul listei [_|T] in loc sa-l dam ca
 * variabila nedenumita: _ , atunci primim avertismentul:
 * variabila singleton. */

concat([],L,L).                      % fapt
concat([H|T],L,[H|M]) :- concat(T,L,M). % regula



                        % Aici: intrebari: formate din scopuri. -------->
```

lung([a,b,c],3).

true

lung([a,b,c],5).

false

lung([a,b,c,d,e],Cat).

Cat = 5

concat([1,2],[3,4,5],[1,2,3,4,5]).

true

concat([1,2],[],[1,2,3,4,5]).

false

concat([a,b,c],[1,2,3,4,5],CeLista).

CeLista = [a, b, c, 1, 2, 3, 4, 5]

?- concat([a,b,c],[1,2,3,4,5],CeLista).

Examples History Solutions table results Run!

Interogări cu scopuri sau variabile multiple

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help 347 users online Search

Program

```
% Aici: fapte si reguli.

lung([],0).                          % fapt: intotdeauna adevarat
lung([_|T],N) :- lung(T,K), N is K+1.   % regula: are structura:
                        % are loc acest fapt :- daca au loc aceste fapte.

/* Daca denumim capul listei [_|T] in loc sa-l dam ca
 * variabila nedenumita: _ , atunci primim avertismentul:
 * variabila singleton. */

concat([],L,L).                      % fapt
concat([H|T],L,[H|M]) :- concat(T,L,M). % regula


% Ati observat sintaxa pentru comentarii:
    % pe un rand
    /* pe mai
     * multe randuri */



% Sa punem si intrebari formate din mai multe scopuri.

% Sa vedem si intrebari care au mai multe raspunsuri.
% Sa cerem cu "Next" mai multe rezultate.
% In loc de "Next", in versiunea desktop a SWI-Prolog, se foloseste ";".
% La final, se afiseaza "false", semnificand ca nu mai exista alte solutii.
```

concat([a,b,c],[1,2,3],Ce),*concat*(Ce,[10,-1],Alta),*lung*(Alta,Cat).

Alta = [a, b, c, 1, 2, 3, 10, -1],
Cat = 8,
Ce = [a, b, c, 1, 2, 3]

concat(CeLista,[1,2,3],[-1,0,1,2,3]).

CeLista = [-1, 0]
false

concat([1,2,3],CuCeLista,[1,2,3,4,5]).

CuCeLista = [4, 5]

concat(CeLista,CuCeLista,[1,2,3]).

CeLista = [],
CuCeLista = [1, 2, 3]
CeLista = [1],
CuCeLista = [2, 3]
CeLista = [1, 2],
CuCeLista = [3]

Next 10 100 1,000 Stop

?- concat(CeLista,CuCeLista,[1,2,3]).

Examples History Solutions table results Run!

Întrebări cu mai multe răspunsuri posibile

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help 340 users online Search

Program

```
% Aici: fapte si reguli.

lung([],0).                          % fapt: intotdeauna adevarat
lung([_|T],N) :- lung(T,K), N is K+1.   % regula: are structura:
                        % are loc acest fapt :- daca au loc aceste fapte.

/* Daca denumim capul listei [_|T] in loc sa-l dam ca
 * variabila nedenumita: _ , atunci primim avertismentul:
 * variabila singleton. */

concat([],L,L).                      % fapt
concat([H|T],L,[H|M]) :- concat(T,L,M). % regula


% Ati observat sintaxa pentru comentarii:
    % pe un rand
    /* pe mai
     * multe randuri */



% Sa punem si intrebari formate din mai multe scopuri.

% Sa vedem si intrebari care au mai multe raspunsuri.
% Sa cerem cu "Next" mai multe rezultate.
% In loc de "Next", in versiunea desktop a SWI-Prolog, se foloseste ";".
% La final, se afiseaza "false", semnificand ca nu mai exista alte solutii.
```

```
concat(CeLista,CuCeLista,[1,2,3]).
CeLista = [],
CuCeLista = [1, 2, 3]
CeLista = [1],
CuCeLista = [2, 3]
CeLista = [1, 2],
CuCeLista = [3]
CeLista = [1, 2, 3],
CuCeLista = []
false

concat([1,2,X,Y],[5,Z,7,T],[1,2,3,4,5,6,7]).
false

concat([1,2,X,Y],[5,Z,7],[1,2,3,4,5,6,7]).
X = 3,
Y = 4,
Z = 6

?-
```

Examples History Solutions table results Run!

Vom vedea cum găsește Prologul aceste răspunsuri

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help 342 users online Search

Program

```
lung([],0).
lung([_|T],N) :- lung(T,K), N is K+1.

concat([],L,L).
concat([H|T],L,[H|M]) :- concat(T,L,M).



% Sa vedem si intrebari care au mai multe raspunsuri.
% Sa cerem cu "Next" mai multe rezultate.
% In loc de "Next", in versiunea desktop a SWI-Prolog, se foloseste ";".
% La final, se afiseaza "false", semnificand ca nu mai exista alte solutii.
```

concat([1,2,X,Y],L,[1,2,3,4,5,6]).

L = [5, 6],
X = 3,
Y = 4

concat([1,2,X,Y|T],L,[1,2,3,4,5,6]).

L = [5, 6],
T = [],
X = 3,
Y = 4

L = [6],
T = [5],
X = 3,
Y = 4

L = [],
T = [5, 6],
X = 3,
Y = 4

false

?- concat([1,2,X,Y|T],L,[1,2,3,4,5,6]).

Examples History Solutions table results Run!

Vedeți și celelalte opțiuni ale SWI–Prolog online

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help

334 users online

Search

Program

```
lung([],0).
lung([_|T],N) :- lung(T,K), N is K+1.

concat([],L,L).
concat([H|T],L,[H|M]) :- concat(T,L,M).

% Am cerut 10 rezultate deodata. ----->
```

L = [1, 2, 3],
M = []

concat(L,M,[1,2,3,4,5]).

L = [],
M = [1, 2, 3, 4, 5]
L = [1],
M = [2, 3, 4, 5]
L = [1, 2],
M = [3, 4, 5]
L = [1, 2, 3],
M = [4, 5]
L = [1, 2, 3, 4],
M = [5]
L = [1, 2, 3, 4, 5],
M = []

concat(L,M,[1,2,3,4,5,6,7]).

L = [],
M = [1, 2, 3, 4, 5, 6, 7]

Next 10 100 1,000 Stop

?- concat(L,M,[1,2,3,4,5,6,7]).

Examples History Solutions table results Run!

Inclusiv opțiuni pentru editare rapidă: vedeți butonul "History"

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help

334 users online

Search

Program

```
lung([],0).
lung([_|T],N) :- lung(T,K), N is K+1.

concat([],L,L).
concat([H|T],L,[H|M]) :- concat(T,L,M).

% Am cerut 10 rezultate deodata. ----->
```

L = [1, 2, 3],
M = []

concat(L,M,[1,2,3,4,5]).

L = [],
M = [1, 2, 3, 4, 5]
L = [1],
M = [2, 3, 4, 5]
L = [1, 2],
M = [3, 4, 5]
L = [1, 2, 3],
M = [4, 5]
L = [1, 2, 3, 4],
M = [5]
L = [1, 2, 3, 4, 5],
M = []

concat(L,M,[1,2,3,4,5,6,7]).

L = [],
M = [1, 2, 3, 4, 5, 6, 7]

Next 10 100 1,000 Stop

?- concat(L,M,[1,2,3,4,5,6,7]).

Examples History Solutions table results Run!

Și acum în versiunea desktop a SWI–Prolog

Am scris cele doua predicate de mai sus într–un fișier text cu extensia *.pl*, care poate fi încărcat În Prologul desktop cu **File** → **Consult** din meniul ferestrei interpretorului Prolog sau folosind comanda (atenție la [ ], apostrofuri și punct): ['unitate_disc:/calea_catre_fisier/nume_fisier.pl'].

Comanda *trace* produce detalierea pașilor urmați de Prolog pentru a rezolva interogările. Renunțare la această afișare detaliată: *notrace*.

trace și săgeată "Step into" în SWI–Prolog online

SWISH -- SWI-Prolog for SHaring

swish.swi-prolog.org

SWISH File Edit Examples Help

129 users online

Search

Program

```
1 lung([],0).
2 lung([_|T],N) :- lung(T,K), N is K+1.
3
4 concat([],L,L).
5 concat([H|T],L,[H|M]) :- concat(T,L,M).
6
```

trace,concat(L,M,[sgl]).

```
Call: concat(_3768, _3764, [sgl])
Exit: concat([], [sgl], [sgl])
L = [],
M = [sgl]
Redo: concat(_3768, _3764, [sgl])
Call: concat(_4448, _3764, [])
Exit: concat([], [], [])
Exit: concat([sgl], [], [sgl])
L = [sgl],
M = []
Redo: concat(_4448, _3764, [])
Fail: concat(_4448, _3764, [])
Fail: concat(_3768, _3764, [sgl])
false
```

trace,concat(L,M,[1,2,3]).

```
Call: concat(_3868, _3864, [1, 2, 3])
Exit: concat([], [1, 2, 3], [1, 2, 3])
```

```
?- trace,concat(L,M,[1,2,3]).
```

Examples History Solutions table results Run!

ENG 12:05 AM 2/14/2020

Tipurile de date în Prolog

Practic, există un *unic tip de date*: **termenii**, cu, câteva *subtipuri*:

- **variabilele**;
- **atomii**: șirurile de caractere speciale și constantele, inclusiv șirurile de caractere cuprinse între apostrofuri, dar excluzând lista vidă [ ] și numerele;
- **numerele** $\begin{cases} \textbf{întregi} \text{ și} \\ \textbf{reale}; \end{cases}$
- **termenii compuși**: $f(arg_1, \ldots, arg_n)$, unde $n \in \mathbb{N}^*$, f este o funcție (și predicatele sunt tot funcții) n–ară (adică de aritate n, cu n argumente), iar $arg_1, \ldots, arg_n$ sunt **termeni**.

Observație (predicate, i.e. cu valori booleene, nu orice fel de funcții)

Faptele constau din predicate.
Regulile sunt formate dintr–un predicat urmat de simbolul neck, urmat de alte predicate unite prin conectori logici.
Interogările sunt formate din predicate prin conectori logici.

Observație (imbricarea)

Sintaxa Prolog permite imbricarea predicatelor cu operații arbitrare, dar imbricarea doar construiește termeni, nu produce, de exemplu, calcule aritmetice în subtermeni (a se vedea mai jos predicatele *e*, *f* din *exlisteetc.pl*).

Operatori Prolog și predicate predefinite în Prolog

Exemplu

Nu putem interoga (pentru că nu este predicat: rezultatul e aritmetic, nu boolean):
?- -((10-1)*2-3)/5.
dar putem interoga:
?- X is -((10-1)*2-3)/5.

Alți operatori aritmetici:	// sau *div*	câtul împărțirii întregi
	mod	restul împărțirii întregi
	**	ridicare la putere

Predicatul *zeroar* (i.e. de aritate 0, i.e. fără argumente) **fail** este întotdeauna evaluat la FALS. Negația: predicatele unare predefinite **not** și \+.

(Predicate unare (i.e. de aritate 1, i.e. cu câte un singur argument))

Predicatele **number**, **integer**, **float** testează dacă argumentul lor este număr, respectiv număr întreg, respectiv număr real (în format float).
Predicatele predefinite **var**, **nonvar**, **atom**, **atomic**, **compound** testează dacă argumentul lor este variabilă, respectiv non–variabilă, respectiv atom, respectiv atom sau număr sau lista vidă [ ], respectiv termen compus.

```
?- var(A).
true.

?- nonvar(A).
false.

?- var(_).
true.

?- nonvar(_).
false.

?- var('A').
false.

?- nonvar('A').
true.

?- var(a).
false.

?- nonvar(a).
true.

?- var(1).
false.

?- nonvar(1).
true.

?- var(p(X)).
false.

?- nonvar(p(X)).
true.

?- var(-->).
false.

?- nonvar(-->).
true.

?- compound([]).
false.

?- compound(X+1).
true.
```

```
?- atom(a).
true.

?- atom(alta_constanta).
true.

?- atom(-#-->).
true.

?- atom(1).
false.

?- atom([]).
false.

?- atom(p(X)).
false.

?- atom(X).
false.

?- atomic(a).
true.

?- atomic(alta_constanta).
true.

?- atomic(-#-->).
true.

?- atomic(1).
true.

?- atomic([]).
true.

?- atomic(p(X)).
false.

?- atomic(X).
false.
```

```
?- number(a).
false.

?- number(10+2).
false.

?- number(12).
true.

?- number(12.5e2).
true.

?- integer(5).
true.

?- integer(-5).
true.

?- integer(5.0).
false.

?- float(5).
false.

?- float(5.0).
true.

?- float(-5.0e2).
true.

?- float(-5e2).
true.

?- float(5e0).
true.

?- float(-15.25e-2).
true.

?- float(-15.25E-2).
true.
```

Pentru a răspunde la o interogare, Prologul încearcă să UNIFICE **scopurile** din **interogare** cu predicatele din:

- **fapte**;
- **membrii stângi** ai **regulilor** – dacă o astfel de unificare reușeste, atunci **următorul scop** pe care va încerca să–l satisfacă este membrul drept al aceleiași reguli, cu argumentele rezultate în urma unificării.

(UNIFICARE (orientativ) – detalii într–un curs următor)

Două funcții UNIFICĂ ddacă: $\begin{cases} \text{acele funcții coincid și} \\ \text{au aceleași argumente:} \end{cases}$

$$f(arg_1, \ldots, arg_n) = g(a_1, \ldots, a_k) \iff \begin{cases} f = g \ (\Longrightarrow n = k) \text{ și} \\ arg_1 = a_1, \\ \vdots \\ arg_n = a_n. \end{cases}$$

Mai precis, două funcții UNIFICĂ ddacă: $\begin{cases} \text{acele funcții coincid și} \\ \text{RECURSIV, } \textbf{argumentele} \text{ lor UNIFICĂ.} \end{cases}$

RECURSIA merge până la **termeni fără argumente**, adică **variabile** sau **constante**:

- **constantele** sunt **operații** (anume exact **operațiile fără argumente** – vom vedea), așadar, conform regulii de mai sus, NU UNIFICĂ DECÂT **cu ele însele și cu orice variabilă**;
- **variabilele** UNIFICĂ, CU **ele însele și cu orice termen care nu le conține**; (desigur, o unificare de tipul $X = h(X)$ ar conduce la $X = h(X) = h(h(X)) = h(h(h(X))) = \ldots$, deci nu ar fi corectă.)

(Operatori pentru unificare, respectiv testarea egalității, inegalității, literal identității: predicate binare infixate predefinite)

```
= testeaza egalitatea ca expresii; face si unificarea, in unele versiuni de Prolog
cu testarea ocurentelor variabilelor in termeni, in alte versiuni fara acest test
unify_with_occurs_check face unificarea cu testarea ocurentelor variabilelor in termeni
\= testeaza nonegalitatea ca expresii
=:= testeaza egalitatea ca valori de expresii aritmetice calculate
=\= testeaza nonegalitatea ca valori de expresii aritmetice calculate
== testeaza literal identitatea
\== testeaza literal nonidentitatea
\+ sau not reprezinta negatia
=< este mai mic sau egal
>= este mai mare sau egal
```

Exemplu (din fișierul *test.pl*)

```
test1 :- 1+5=3+3.
test2 :- 1+5\=3+3.
test3 :- 1+5=:=3+3.
test4 :- 1+5=\=3+3.

/*
Dati interogarile:
?- test1.
...
?- test4.
sau, direct:
?- 1+5=3+3.
...
?- 1+5=\=3+3.
Dati interogarile:
?- X==c.
?- X==Y.
?- X==X.
?- X\==c.
?- X\==c.
?- X\==Y.
?- X\==X.
Observati ca nu se unifica variabila X cu constanta c sau cu variabila Y,
ci se testeaza daca membrii egalitatii sunt literal identici.
```

Dupa cum am observat in exemplele cu predicatul *concat* de mai sus:

```
Se pot da si interogari de forma: predicat(Var1,ct1,Var2,_,ct2,_,Var3,...).
Atunci, pentru fiecare satisfacere a lui predicat, se afiseaza doar
valorile lui Var1, Var2, Var3,... care satisfac predicatul.*/
```

```
test5 :- X=f(V,g(X)).
test6 :- unify_with_occurs_check(X,f(V,g(X))).
test7 :- f(X,g(X))=f(a,Y).
test8 :- f(X,g(X))=f(X,Y).
test9 :- f(X,g(X))=f(X,a).
test10 :- f(X,g(X))=f(g(X),Y).
test11 :- unify_with_occurs_check(f(X,g(X)),f(g(X),Y)).
test12 :- f(X,g(a))=f(Y,Y).
test13 :- f(X,g(X))=f(Y,Y).
test14 :- unify_with_occurs_check(f(X,g(X)),f(Y,Y)).

/*
Daca dam interogarile:
?- test5.
...
?- test14.
interogari in care nu apar variabile, atunci raspunsurile Prologului vor fi true sau false.
Daca dam, ca interogari, doar:
?- X=f(V,g(X)).
?- unify_with_occurs_check(X,f(V,g(X))).
?- f(X,g(X))=f(a,Y).
?- f(X,g(X))=f(X,Y).
?- f(X,g(X))=f(X,a).
?- f(X,g(X))=f(g(X),Y).
?- unify_with_occurs_check(f(X,g(X)),f(g(X),Y)).
?- f(X,g(a))=f(Y,Y).
?- f(X,g(X))=f(Y,Y).
?- unify_with_occurs_check(f(X,g(X)),f(Y,Y)).
atunci afiseaza si unificatorul, daca acesta exista.
```

Dupa cum am vazut mai sus:

```
Afisarea urmatoarelor instante ale variabilelor care satisfac predicatul: ;
sau, in SWISH SWI-Prolog-ul online: NEXT, sau numarul de urmatoare instante de afisat.
*/
```

Exemplu (imbricare: din fișierul *eximbric.pl*)

```
% Putem da astfel de fapte sau reguli:

c(ct,_).

d(X,c(a,X)).
d(X,c(V,Y)) :- d(X,c(V,X)) ; c(V,Y).  %% "," inseamna conjunctie logica ("si"), ";" semnifica disjunctie logica ("sau")

% Dar, de exemplu, in urmatorul fapt, "-" este privit ca un operator binar care construieste termeni:

e(_-_).

/* La fiecare dintre interogarile:
?- e(5-2).
?- e(a-f(X)).
?- e((a-b)-c).
Prologul raspunde true, dar la interogarea:
?- e(3).
Prologul raspunde false, pentru ca termenul X-Y, de operator dominant "-", si constanta 3 nu unifica. */

f(0) :- write(nope).
f(_-1) :- write(yap).
f(X) :- X>0, write('Hello, world!'), tab(1), f(X-1).

/* Dati interogarea:
?- f(1).
*/
```

Exemplu (supraîncărcare: din fișierul *ex_supraincarcare.pl*)

```
p(a).
p(p). % p este si predicat unar, si constanta

p(a,b). % p este si predicat binar
p(p,1).
p(X,Y) :- p(X), p(Y).
```

(Lista vidă nu unifică, cu nicio listă nevidă)

Lista vidă [ ] e o constantă, i.e. operație fără argumente (vom vedea), iar o listă nevidă $[H|T]$ (care conține măcar elementul H) este termenul format din operația binară [|] cu argumentele H și T ([|](H, T)). Cum operațiile [ ] și [|] nu coincid, rezultă că [ ] nu unifică, cu $[H|T]$.

Exemplu (utilitate *var*, *nonvar*: din fișierul *exspllliste.pl*)

Să numărăm aparițiile unui element într–o listă, cu un predicat ternar având argumentele: *Elementul*, *Lista*, *Nr*, care va calcula numărul de apariții în acest al treilea argument.
Următoarea implementare numără, de fapt, elementele listei cu care unifică acel element:

```
aparitii(_,[],0).
aparitii(X,[X|T],N) :- aparitii(X,T,K), N is K+1.
aparitii(X,[H|T],N) :- X\=H, aparitii(X,T,N).
```

Iată răspunsul Prologului la această interogare:

```
?- aparitii(a,[X,a,1,a,a,2,3,Y,a,4,Z],Nr).
X = Y, Y = Z, Z = a,
Nr = 7 ;
false.
```

Am cerut mai multe soluții, cu ";", și răspunsul Prologului a fost "false", așadar acesta e singurul răspuns găsit de Prolog la interogarea de mai sus.

În schimb, dacă implementăm astfel predicatul de mai sus:

```
nr_aparitii(_,[],0).
nr_aparitii(X,[H|T],N) :- var(H), nr_aparitii(X,T,N).
nr_aparitii(X,[H|T],N) :- nonvar(H), X=H, nr_aparitii(X,T,K), N is K+1.
nr_aparitii(X,[H|T],N) :- X\=H, nr_aparitii(X,T,N).
```

atunci răspunsul unic găsit de Prolog la aceeași interogare este:

```
?- nr_aparitii(a,[X,a,1,a,a,2,3,f(X),a,4,Z],Nr).
Nr = 4 .
```

Iată, în schimb, răspunsul Prologului la această interogare:

```
?- nr_aparitii(f(V),[X,a,1,a,a,2,3,f(X),a,4,Z],Nr).
V = X,
Nr = 1 ;
false.
```

Putem număra termenii literal identici cu cel căutat în listă astfel:

```
numar_aparitii(_,[],0).
numar_aparitii(X,[H|T],N) :- X\==H, numar_aparitii(X,T,N).
numar_aparitii(X,[H|T],N) :- X==H, numar_aparitii(X,T,K), N is K+1.
```

Iată cum funcționează această implementare:

```
?- numar_aparitii(a,[X,a,1,a,a,2,3,f(X),a,4,Z],Nr).
Nr = 4 .

?- numar_aparitii(X,[X,a,1,a,a,2,3,f(X),a,4,Z],Nr).
Nr = 1 .

?- numar_aparitii(V,[X,a,1,a,a,2,3,f(X),a,4,Z],Nr).
Nr = 0 .

?- numar_aparitii(f(V),[X,a,1,a,f(X),a,f(V),2,f(X),3,f(Y),f(V),a,4,Z],Nr).
Nr = 2 .
```

Cum răspunde Prologul la interogări?

- Considerăm baza de cunoștințe:

```
concat([ ], L, L).
concat([H|T], L, [H|M]) :- concat(T, L, M).
```

După cum am văzut mai sus, predicatul ternar $concat(L1, L2, L)$ e satisfăcut ddacă variabilele $L1$, $L2$, L iau ca valori liste cu proprietatea că lista L se obține prin concatenarea listei $L1$ cu lista $L2$.

- Să vedem cum răspunde Prologul la interogarea:

```
?- concat(Ce,CuCe,[1,2,3]).
```

adică:

ce valori ale variabilelor Ce și $CuCe$ satisfac predicatul $concat(Ce, CuCe, [1, 2, 3])$?

Adică: pentru ce liste Ce, $CuCe$ are loc proprietatea că lista $[1, 2, 3]$ este rezultatul concatenării listei Ce cu lista $CuCe$?

Următoarea regulă este valabilă și pentru **scopuri compuse**, adică formate din *mai multe scopuri* constând din *cate un predicat*, legate prin conjuncție (virgulă), disjuncție (punct și virgulă) etc., în cazul cărora scopurile sunt satisfăcute pe rând, iar *valorile variabilelor* care apar în acele scopuri trebuie să fie aceleași pentru toate scopurile, în răspunsul la o **interogare** constând din acel **scop compus**.

Pentru a satisface un **scop**, Prologul caută să unifice termenul dat de predicatul din acel scop cu:

- un **fapt**, caz în care răspunsul la o **interogare** constând din acel scop este: **true**, dacă scopul nu conține variabile sau e satisfăcut pentru orice valori ale variabilelor care apar în el;
- **membrul stâng** al unei **reguli**, caz în care următorul **scop** (care poate fi un **scop compus**) este dat de membrul drept al acelei reguli în care variabilele sunt înlocuite cu termenii pentru care s–a realizat unificarea între **scopul** curent și **membrul stâng** al acelei **reguli**.

De exemplu, pentru interogarea de mai sus:

Vom vedea într–un curs următor ce este un *unificator*. Grosso modo:

un **unificator** constă în *valori* ale *variabilelor* care apar în doi termeni pentru a avea loc *unificarea* celor doi termeni; respectivele *valori* ale *variabilelor* sunt date de alți *termeni*.

Orice *unificator* este o *substituție*. Grosso modo, o *substituție* este o funcție care dă variabilelor ca valori termeni.
Dacă $n \in \mathbb{N}$, $V_1, \ldots, V_n$ sunt variabile, iar $T_1, \ldots, T_n$ sunt termeni, atunci notăm cu

$$\{V_1/T_1, \ldots, V_n/T_n\}$$

substituția care dă ca valori variabilelor $V_1, \ldots, V_n$, respectiv, termenii $T_1, \ldots, T_n$.
Substituțiile se extind de la variabile la toți termenii, așadar se pot compune ca funcții, obținându–se tot substituții.
Pentru un termen arbitrar T, valoarea $\{V_1/T_1, \ldots, V_n/T_n\}(T)$ a substituției $\{V_1/T_1, \ldots, V_n/T_n\}$ în termenul T se obține prin înlocuirea, în T, a variabilelor $V_1, \ldots, V_n$, respectiv, cu termenii $T_1, \ldots, T_n$; desigur, se vor face înlocuirile doar pentru acelea dintre variabilele $V_1, \ldots, V_n$ care apar în T.

Dacă variabilele care apar doi termeni, T și U, sunt $V_1, \ldots, V_n$ pentru un $n \in \mathbb{N}$, atunci un unificator pentru T și U este o substituție $\{V_1/T_1, \ldots, V_n/T_n\}$ care ia aceeași valoare în T și U:

$$\{V_1/T_1, \ldots, V_n/T_n\}(T) = \{V_1/T_1, \ldots, V_n/T_n\}(U).$$

Când unifică termenii T și U, Prologul obține un *cel mai general unificator*, adică un unificator $\{V_1/T_1, \ldots, V_n/T_n\}$ cu proprietatea că orice unificator al termenilor T și U se obține prin înlocuirea variabilelor din termenii $T_1, \ldots, T_n$ cu termeni, adică prin compunerea unei substituții arbitrare cu unificatorul $\{V_1/T_1, \ldots, V_n/T_n\}$.

Exemplu

Un cel mai general unificator pentru termenii $concat([H|T], L, [H|M])$ și $concat([1,2,3],[4,5], Lista) = concat([1|[2,3]],[4,5], Lista)$ este:

$$\{H/1,\ T/[2,3],\ L/[4,5],\ Lista/[1|M]\}$$

Un alt unificator pentru aceiași termeni este, de exemplu, $\{H/1,\ T/[2,3],\ L/[4,5],\ M/[a,b],\ Lista/[1,a,b]\}$, dat de compunerea unificatorului de mai sus cu substituția $\{M/[a,b]\}$.

Dacă dăm *trace* și apoi interogarea:

```
?- concat(Ce,CuCe,[a,b,c]).
```

observăm din pașii afișați că primul lucru executat de Prolog este redenumirea variabilelor din interogare în nume de forma *_număr*.
Dacă în baza de cunoștințe, avem și definițiile altor predicate, de exemplu definiția de mai sus a predicatului lung, atunci:

cum $\mathrm{concat} \neq \mathrm{lung}$, scopul $\mathrm{concat}(\mathrm{Ce}, \mathrm{CuCe}, [\mathrm{a}, \mathrm{b}, \mathrm{c}])$ nu unifică:

- nici cu $\mathrm{lung}([\,], 0)$,
- nici cu $\mathrm{lung}([\mathrm{H}|\mathrm{T}], \mathrm{N})$.

În schimb, concat(Ce,CuCe,[1,2,3]) unifică:

cu $\mathrm{concat}([\,], \mathrm{L}, \mathrm{L})$, pentru $\mathrm{Ce} = [\,]$ și $\mathrm{CuCe} = \mathrm{L} = [1, 2, 3]$ – aceasta este *prima soluție* a interogării: *unificatorul* $\{Ce/[\,], CuCe/[1, 2, 3], L/[1, 2, 3]\}$,

dar și:

cu $\mathrm{concat}([\mathrm{H}|\mathrm{T}], \mathrm{L}, [\mathrm{H}|\mathrm{M}])$, pentru $\mathrm{H} = 1$, $\mathrm{Ce} = [1|\mathrm{T}]$, $\mathrm{CuCe} = \mathrm{L}$ și $\mathrm{M} = [2, 3]$,

așadar **următorul scop** este membrul stâng $concat(T, L, M)$ al acestei reguli, cu variabilele înlocuite cu aceste valori, adică unificatorul de mai sus aplicat membrului stâng al acestei reguli:
$\{H/1, Ce/[1|T], L/CuCe, M/[2,3]\}(concat(T, L, M)) = concat(T, CuCe, [2,3])$:

?- concat(T,CuCe,[2,3]).

Acesta unifică:

cu $\mathrm{concat}([\,], \mathrm{L}, \mathrm{L})$, pentru $\mathrm{T} = [\,]$ și $\mathrm{CuCe} = \mathrm{L} = [2,3]$ – așadar *a doua soluție* a interogării este: $\mathrm{Ce} = [1|\mathrm{T}] = [1|[\,]] = [1]$ și $\mathrm{CuCe} = [2,3]$, adică unificatorul $\{Ce/[1], CuCe/[2,3]\}$,

dar și:

cu $\mathrm{concat}([\mathrm{H1}|\mathrm{T1}], \mathrm{L}, [\mathrm{H1}|\mathrm{M1}])$, pentru $\mathrm{H1} = 2$, $\mathrm{T} = [2|\mathrm{T1}]$, $\mathrm{CuCe} = \mathrm{L}$, $\mathrm{M1} = [3]$,

așadar **următorul scop** este:

?- concat(T1,CuCe,[3]).

Acesta unifică:

cu $\mathrm{concat}([\,],\mathrm{L},\mathrm{L})$, pentru T1 = [ ] și CuCe = L = [3] – așadar *a treia soluție* a interogării este: Ce = [1|T] = [1|[2|T1]] = [1|[2|[ ]]] = [1, 2] și CuCe = [3],

dar și:

cu $\mathrm{concat}([\mathrm{H2}|\mathrm{T2}],\mathrm{L},[\mathrm{H2}|\mathrm{M2}])$, pentru H2 = 3, T1 = [3|T2], CuCe = L și M2 = [ ],

așadar **următorul scop** este:

?- concat(T2,CuCe,[ ]).

Întrucât:

constanta [ ] nu unifică, cu termenul $[\mathrm{H3}|\mathrm{M3}]$, având operația dominantă [|],

avem:

$\mathrm{concat}(\mathrm{T2}, \mathrm{CuCe}, [\,])$ nu unifică, cu $\mathrm{concat}([\mathrm{H3}|\mathrm{T3}], \mathrm{L}, [\mathrm{H3}|\mathrm{M3}])$.

Prin urmare:

$\mathrm{concat}(\mathrm{T2}, \mathrm{CuCe}, [\,])$ unifică numai cu $\mathrm{concat}([\,], \mathrm{L}, \mathrm{L})$, pentru $\mathrm{T2} = [\,]$ și $\mathrm{CuCe} = \mathrm{L} = [\,]$ – așadar *a patra* și *ultima soluție* a interogării este:
$\mathrm{Ce} = [1|\mathrm{T}] = [1|[2|\mathrm{T1}]] = [1|[2|[3|\mathrm{T2}]]] = [1|[2|[3|[\,]]]] = [1, 2, 3]$ și $\mathrm{CuCe} = [\,]$.

• În cazul unui scop compus, de exemplu:

```
?- concat(A,[2,3],[1,2,3]),concat(A,[4,5],C).
```

toate scopurile trebuie satisfăcute, cu **aceleași valori** pentru *variabilele comune*.
Pentru acest exemplu, *unica soluție* este: $\mathrm{A} = [1]$, $\mathrm{C} = [1, 4, 5]$.

La fel se caută soluțiile pentru predicatele care produc nu numai unificări de termeni, ci și calcule, ca, de exemplu, predicatul pentru calculul lungimii unei liste, afișări pe ecran, de exemplu cu predicatul unar predefinit *write*, care scrie argumentul său pe ecran, apoi intoarce **true**, etc..

- De exemplu, pentru baza de cunoștințe:

```
lung([],0).
lung([_|T],N) :- lung(T,K), N is K+1.
```

și interogarea:

```
?- lung([1,2,3],NrElem).
```

Scopul din această interogare unifică doar cu membrul stâng al regulii din această bază de cunoștințe:

Așadar următorul scop este scopul compus:

```
?- lung([2,3],K), N is K+1.
```

Primul dintre aceste două scopuri unifică tot numai cu membrul stâng al regulii:

Următorul scop este:

```
?- lung([3],K1), N1 is K1+1.
```

din cadrul căruia primul scop unifică numai cu membrul stâng al regulii:

Următorul scop este acest scop compus, din cadrul căruia primul scop unifică numai cu faptul:

```
?- lung([ ],K2), N2 is K2+1.
```

Unificarea $K2 = 0$ conduce la: $N2 = 1$, pentru că $N2$ is $0 + 1$ produce unificarea $N2 = 1$, apoi întoarce **true**, și se continuă în acest mod la revenirea din pașii acestui backtracking: $K1 = N2 = 1$, $K = N1 = 2$ $(K1 + 1)$, $NrElem = N = 3$ $(K + 1)$. Așadar *unica soluție* a interogării de mai sus este: $NrElem = 3$.

Să reținem că predicatul predefinit binar infixat **is** produce calculul expresiei aritmetice din argumentul său drept, apoi unificarea între valoarea obținută din acest calcul și argumentul său stâng.
Pentru aceeași bază de cunoștințe ca mai sus, să vedem cum răspunde Prologul la interogarea:

```
?- lung(CeLista,3).
```

Unificarea $0 = 3$ nu are loc, așadar acest scop nu unifică, cu faptul, dar unifică, cu membrul stâng al regulii din această bază de cunoștințe:

Rezultatul acestei unificări este: CeLista=[H|T] și N=3, așadar următorul scop este:

```
?- lung(T,K), 3 is K+1.
```

Primul termen al conjuncției care dă acest scop compus unifică atât cu faptul, cât și cu membrul stâng al regulii:

Prima dintre aceste unificări, având rezultatul T=[] și K=0, produce satisfacerea scopului lung(T,K) (fiind unificare cu un fapt). Prolog–ul trece apoi la satisfacerea membrului drept al conjuncției de mai sus, cu această valoare pentru K: 3 is 0+1, conducând la unificarea 3=1, care eșuează.
Rezultatul celei de–a doua dintre aceste unificări este: T=[H1|T1] și K=N1, așadar următorul scop este:

```
?- lung(T1,K1), N1 is K1+1.
```

Primul termen al acestei conjuncții unifică atât cu faptul, cât și cu membrul stâng al regulii:

Prima dintre aceste unificări, având rezultatul T1=[] și K1=0, produce satisfacerea scopului lung(T1,K1). Prolog–ul trece apoi la satisfacerea membrului drept al acestei conjuncții, cu această valoare pentru K1: N1 is 0+1, care conduce la unificarea N1=1. Unificarea anterioară devine: T=[H1|[]]=[H1] și K=N1=1, iar scopul anterior devine: lung([H1],1), 3 is 1+1. Membrul drept al acestei conjuncții duce la unificarea 3=2, care eșuează.
A doua dintre aceste unificări, având rezultatul T1=[H2|T2] și K1=N2, conduce la scopul:

?- lung(T2,K2), N2 is K2+1.

Primul termen al acestei conjuncții unifică atât cu faptul, cât și cu membrul stâng al regulii:

Acest scop este satisfăcut cu prima dintre aceste unificări: T2=[] și K2=0, urmată de N2 is 0+1, care produce unificarea N2=1. Așadar scopul lung(T1,K1), N1 is K1+1 este satisfăcut cu unificarea: T1=[H2|[ ]]=[H2] și K1=N2=1, urmată de N1 is 1+1, care produce unificarea N1=2.
Prolog–ul satisface apoi scopul lung(T,K), 3 is K+1 cu unificarea: T=[H1|[H2]]=[H1,H2] și K=N1=2, urmată de 3 is 2+1, care produce unificarea 3=3. Iar scopul inițial, lung(CeLista,3), este acum satisfăcut cu unificarea: CeLista=[H|[H1,H2]]=[H,H1,H2].
Dacă mai cerem soluții, Prolog–ul va efectua și unificarea T2=[H3|T3] și K2=N3, urmată de scopul compus lung(T3,K3), N3 is K3+1. Ca mai sus, unificarea scopului lung(T3,K3) cu faptul din baza de cunoștințe conduce la T=[H1,H2,H3] și 3 is 3+1, care produce unificarea 3=4. Aceasta eșuează, așadar Prolog–ul va aplica regula din baza de cunoștințe pentru satisfacerea scopului lung(T3,K3). Urmând pașii de mai sus, se va ajunge la T=[H1,H2,H3,H4] și 3 is 4+1, care produce unificarea 3=5, care eșuează. Toate aceste următoare unificări: 3=6, 3=7 ș.a.m.d. eșuează, până la umplerea stivei de lucru.

- Acum să considerăm următoarea bază de cunoștințe:

```
concat([],L,L).                                          % (f0)
concat([H|T],L,[H|M]) :- concat(T,L,M).                  % (r0)

inversa([],[]).                                          % (f1)
inversa([H|T],L) :- inversa(T,M), concat(M,[H],L).       % (r1)
```

și interogarea:

```
?- inversa([1,2,3],CeLista).
```

Cum $[1,2,3] = [1|[2,3]]$ nu unifică, cu [ ], scopul $inversa(L,[1,2,3])$ unifică doar cu membrul stâng al regulii ($r1$):

Așadar următorul scop este scopul compus:

```
?- inversa([2,3],M), concat(M,[1],CeLista).
```

Ca mai sus, Prologul caută să satisfacă mai întâi primul dintre aceste 2 scopuri,

care unifică tot numai cu membrul stâng al regulii ($r1$):

Următorul scop este dat de membrul stâng al regulii ($r1$) pentru variabilele de mai sus, legat prin conjuncție de al doilea dintre scopurile din scopul compus anterior:

```
?- inversa([3],M1), concat(M1,[2],M), concat(M,[1],CeLista).
```

Primul dintre aceste 3 scopuri unifică tot numai cu membrul stâng al regulii ($r1$):

Următorul scop compus:

```
?- inversa([],M2), concat(M2,[3],M1), concat(M1,[2],M), concat(M,[1],CeLista).
```

Primul dintre aceste 4 scopuri unifică cu faptul ($f1$):

După această unificare, noul scop compus este:

```
?- concat([],[3],M1), concat(M1,[2],M), concat(M,[1],CeLista).
```

Primul dintre aceste 3 scopuri unifică, cu faptul ($f0$):

După această nouă unificare, noul scop compus este:

```
?- concat([3],[2],M), concat(M,[1],CeLista).
```

Cum $[3] = [3|[\,]]$, care nu unifică, cu $[\,]$, primul dintre aceste scopuri unifică doar

cu membrul stâng al regulii ($r0$):

După această unificare, noul scop compus este:

```
?- concat([],[2],M3), concat([3|M3],[1],CeLista).
```

Primul dintre aceste 2 scopuri unifică cu faptul ($f0$):

După această unificare, noul scop este:

```
?- concat([3,2],[1],CeLista).
```

Acesta unifică, cu membrul stâng al regulii ($r0$):

Următorul scop este:

```
?- concat([2],[1],M4).
```

Acesta unifică tot cu membrul stâng al regulii ($r0$):

Următorul scop este:

```
?- concat([],[1],M5).
```

Acesta unifică cu faptul ($f0$):

Așadar *unica soluție* a interogării de mai sus este:
$CeLista = [3|M4] = [3|[2|M5]] = [3|[2|[1]]] = [3, 2, 1]$.

- Acum să considerăm baza de cunoștințe precedentă și interogarea:

```
?- inversa(CeLista,[1,2,3]).
```

Cum $[1, 2, 3]$ nu unifică, cu $[\,]$, scopul din această interogare unifică doar cu membrul stâng al regulii ($r1$):

Așadar următorul scop este scopul compus:

```
?- inversa(T,M), concat(M,[H],[1,2,3]).
```

Primul dintre aceste două scopuri unifică atât cu faptul ($f1$), cât și cu membrul stâng al regulii ($r1$):

($f1$) fiind un fapt, care spune, așadar, că *inversa*([ ], [ ]) e satisfăcut, adică întoarce **true**,

pentru această primă unificare nu mai rămâne de satisfăcut decât scopul:

```
?- concat([ ],[H],[1,2,3]).
```

Acesta nu unifică, cu membrul stâng al regulii ($r0$), pentru că [ ] nu unifică, cu o listă nevidă, și nu unifică nici cu faptul ($f0$), așadar nu este satisfăcut, adică întoarce **false**:

Prologul trece la a doua unificare de mai sus, așadar următorul scop este:

```
?- inversa(T1,M1), concat(M1,[H1],M), concat(M,[H],[1,2,3]).
```

Ca mai sus, scopul *inversa*($T1, M1$) unifică:

① cu faptul $(f1)$, pentru $T1 = M1 = [\,]$, iar, cu această unificare, următorul scop este:

```
?- concat([ ],[H1],M), concat(M,[H],[1,2,3]).
```

primul dintre aceste două scopuri unifică numai cu faptul $(f0)$, pentru $M = [H1]$, astfel că următorul scop va fi:

```
?- concat([H1],[H],[1,2,3]).
```

acesta unifică doar cu membrul stâng $concat([H2|T2], L2, [H2|M2])$ al regulii $(r0)$, cu $H1 = H2 = 1$, $T2 = [\,]$, $L2 = [H]$ și $M2 = [2, 3]$ iar, după această unificare, următorul scop este:

```
?- concat([ ],[H],[2,3]).
```

ca mai sus, scopul $concat([\,], [H], [2, 3])$ nu unifică nici cu faptul $(f0)$, nici cu membrul stâng al regulii $(r0)$;

② cu membrul stâng $inversa([H2|T2], L2)$ al regulii $(r1)$, pentru $T1 = [H2|T2]$ și $M1 = L2$, iar, cu această unificare, următorul scop este:

```
?- inversa(T2,M2), concat(M2,[H2],M1), concat(M1,[H1],M),
concat(M,[H],[1,2,3]).
```

Scopul $inversa(T2, M2)$ unifică:

ⓘ cu faptul $(f1)$, pentru $T2 = M2 = [\,]$, iar, pentru această unificare, următorul scop este:

```
?- concat([ ],[H2],M1), concat(M1,[H1],M), concat(M,[H],[1,2,3]).
```

scopul $concat([\,], [H2], M1)$ unifică numai cu faptul $(f0)$, pentru $M1 = [H2]$, așadar următorul scop este:

```
?- concat([H2],[H1],M), concat(M,[H],[1,2,3]).
```

aceste scopuri se rezolvă în aceeași manieră backtracking, obținându–se unica soluție: $H = 3$, $M = [1, 2]$, $H1 = 2$ și $H2 = 1$, așadar obținem *soluția*: $CeLista = [H|T] = [H|[H1|T1]] = [H|[H1|[H2|T2]]] = [3|[2|[1|[\,]]]] = [3, 2, 1]$;
(ii) cu membrul stâng $inversa([H3|T3], M')$ al regulii $(r1)$, pentru $T2 = [H3|T3]$ și $M' = M2$, așadar, dacă mai cerem soluții după cea de mai sus, Prologul trece la scopul:

```
?- inversa(T3,M3), concat(M3,[H3],M2), concat(M2,[H2],M1),
concat(M1,[H1],M), concat(M,[H],[1,2,3]).
```

Acest scop compus nu are soluții, pentru că, conjuncția ultimelor 4 dintre aceste 5 scopuri nu are soluții, dar recurența continuă până se umple stiva de lucru a Prologului, spre deosebire de interogarea anterioară:

```
?- inversa(CeLista,[1,2,3]).
```

care are aceeași unică soluție, dar după obținerea căreia backtracking–ul executat de Prolog se încheie, pentru că nu mai există unificări posibile de efectuat.

Ordinea aplicării clauzelor din baza de cunoștințe

În cazul în care un scop unifică, cu mai multe fapte sau membri stângi de reguli, clauzele din definiția predicatului din acel scop sunt aplicate în ordinea apariției acestora în baza de cunoștințe (de sus în jos).

Exemplu (din fișierul *exsplgraf.pl*)

Observați că predicatul binar *arc* definește un graf orientat care conține circuite:

```
arc(1,2).
arc(1,3).
arc(2,3).
arc(2,4).
arc(3,3).
arc(3,4).
arc(4,1).

drum(V,W) :- arc(V,W).
drum(V,W) :- arc(V,U), drum(U,W).
```

Predicatul binar $drum(V, W)$ e satisfăcut ddacă există drum de la vârful V la vârful W în acest graf, astfel că, la interogarea: `?- drum(1,4).`, Prologul răspunde **true**. Dar, dacă interschimbăm cele două clauze din definiția predicatului *drum*, punând regula înaintea faptului, atunci aceeași interogare produce ciclare, pentru că Prologul va căuta mai întâi un nod U astfel încât să existe un arc de la 1 la U și un drum de la U la 4, apoi va proceda la fel pentru satisfacerea scopului $drum(U, 4)$ și va continua astfel, intrând în unul dintre circuitele din acest graf; dacă graful nu ar avea circuite (între nodurile pentru care dăm interogarea), atunci ordinea clauzelor din definiția predicatului *drum* nu ar avea importanță.

Predicatul *cut* (!)

Observați, mai jos, predicatul zeroar (adică fără argumente) *fail*, care întotdeauna eșuează, adică întotdeauna e evaluat la **false**, și predicatul unar predefinit *not* sau \+, care primește drept argument un predicat și întoarce **true** când acel predicat eșuează și **false** când acel predicat e satisfăcut.
După cum am văzut mai sus, pentru satisfacerea unui scop, faptele și regulile sunt aplicate în ordinea în care sunt scrise în baza de cunoștințe, printr–un backtracking încorporat în interpretorul Prologului.

Predicatul predefinit *cut* ("!") are funcția de a tăia backtrackingul executat pentru satisfacerea unui scop dintr-o interogare, astfel că, după ce scopul respectiv e satisfăcut prin aplicarea regulii în care apare *cut*, nu se mai caută alte satisfaceri ale acelui scop.

Exemplu (din fișierul *testcut.pl*)

Predicatele următoare testează apartenența unui element la o listă:

```
membru(_,[]) :- fail. %%% corect si: not(membru(_,[])).
membru(H,[H|_]).
membru(X,[_|T]) :- membru(X,T).

apartine(_,[]) :- fail. %%% corect si: not(apartine(_,[])).
apartine(H,[H|_]) :- !.
apartine(X,[_|T]) :- apartine(X,T).
```

(Implementarea lui *not* folosind *cut* și *fail* și a unui predicat similar lui *var* folosind *not*; generarea tuturor soluțiilor unui scop utilizând *fail*)

```
/* Dati interogarile:
?- membru(a,[X,a,1,Y,a,a,2,3]).
?- apartine(a,[X,a,1,Y,a,a,2,3]).
si cereti toate variantele de satisfacere (cu ";"/"Next"). */
```

Observați că primul scop de mai sus e satisfăcut de atâtea ori câte apariții are acel element în acea listă.

Implementarea predicatului predefinit *not* este:

```
not(P) :- P, !, fail.  /* daca P e adevarat, atunci taie backtrackingul si esueaza,
astfel ca renunta si la eventualele unificari pentru care a fost satisfacut P) */
not(_).                % in caz contrar, deci cand P e fals, e satisfacut
```

Iar acest predicat poate fi folosit pentru a implementa un predicat care funcționează precum cel predefinit *var*, având în vedere că un termen e variabilă ddacă acel termen unifică cu două constante diferite, dar, bineînțeles, cele două unificări nu au loc în același timp, și nu dorim ca satisfacerea acestui predicat să se realizeze în urma unei unificări a argumentului său cu o constantă:

```
variabila(X) :- not(not(X=a)), not(not(X=b)).
```

Predicatul *fail* poate fi folosit pentru generarea tuturor soluțiilor unui scop, ca în acest exemplu pentru scopul $membru(X, L)$:

```
afis_toti_membrii(L) :- membru(X,L), write(X), tab(1), fail.
```

Folosim *cut* pentru a nu repeta evaluări care consumă timp

Exemplu (din fișierul *testordsatconjdisj.pl*)

```
% prim(N)=true <=> N e numar natural prim

prim(0) :- fail.
prim(1) :- fail.
prim(N) :- integer(N), N>1, nusediv(N,2).

/* nusediv(N,D)=true <=> N nu se divide cu niciun numar natural >= D si
                                    =< decat radacina patrata a lui N */

nusediv(N,D) :- C is D*D, (C>N; C=<N, N mod D>0, Div is D+1, nusediv(N,Div)).

% selectprime(L,LP)=true <=> LP = lista numerelor naturale prime din lista L

selectprime([],[]).
selectprime([H|T],[H|M]) :- prim(H), selectprime(T,M), !.
selectprime([H|T],M) :- selectprime(T,M).

/* Nu doresc sa repet evaluarea predicatului prim(H), intrucat consuma timp,
asa ca dau "cut". Prologul trece la ultima regula doar daca H nu e prim. */
```

Ordinea satisfacerii predicatelor în conjuncții/disjuncții

Într–o conjuncție (",") sau disjuncție (";") de predicate, Prologul face evaluarea acelor predicate de la stânga la dreapta, și:

- iese dintr–o conjuncție când întâlnește un predicat evaluat la **false**,
- iese dintr–o disjuncție când întâlnește un predicat evaluat la **true**,

adică nu evaluează și predicatele de la dreapta acelui predicat în acea conjuncție/disjuncție.

Exemplu (din fișierul *testordsatconjdisj.pl*)

```
f(X) :- X>10, write(X), X<0.

f1(X) :- X>10, write(X), tab(1), Y is X-1, f1(Y), X<0.

g(X) :- X>10, X<0, write(X).

g1(X) :- X>10, write(X), X<0, tab(1), Y is X-1, g1(Y).

h(X) :- X<0, write('negativ'); integer(X), X mod 2 =:= 0, write('par'); write('nope').

h1(X) :- X<0, write('negativ'), !; integer(X), X mod 2 =:= 0, write('par'), !; write('nope').

k(X) :- integer(X), X mod 2 =:= 0, write('par'); X<0, write('negativ'); write('nope').

k1(X) :- integer(X), X mod 2 =:= 0, write('par'), !; X<0, write('negativ'), !; write('nope').
```

```
/* Interogati, dand ";"/"Next" pentru apelurile lui h, k:
?- f(100).   % e evaluat la false dupa scrierea lui 100 pe ecran
?- g(100).   % e evaluat la false si nu se mai scrie 100 pe ecran
?- f1(100).  % e evaluat la false dupa apelul recursiv
?- g1(100).  % e evaluat la false si nu se mai face apelul recursiv
?- h(-100).
?- k(-100).
?- h(100).
?- k(100).
?- h(99).
?- k(99).
% Dupa satisfacerea disjunctiei, se evalueaza restul acesteia doar daca mai cerem solutii
?- h1(-100). % are o singura solutie, datorita lui cut
?- k1(-100). % are o singura solutie, datorita lui cut */
```

Remarcă (scriere echivalentă pentru disjuncții)

```
% Implementare echivalenta pentru predicatul h de mai sus:

hv(X) :- X<0, write('negativ').
hv(X) :- integer(X), X mod 2 =:= 0, write('par').
hv(X) :- write('nope').

/* Analog, o regula de urmatoarea forma - in care parantezele sunt necesare,
spre deosebire de regulile anterioare, intrucat  conjunctia (",") are prioritatea
mai mare decat a disjunctiei (";") -, regula care spune ca m(X) e satisfacut daca
X e numar natural care e sau multiplu de 3, sau multiplu de 5, si astfel incat
X+1 e multiplu de 7: */

m(X) :- integer(X), X>=0, (X mod 3 =:= 0 ; X mod 5 =:= 0), Y is X+1, Y mod 7 =:= 0.

% poate fi scrisa echivalent sub forma urmatoarelor doua reguli:

mv(X) :- integer(X), X>=0, X mod 3 =:= 0, Y is X+1, Y mod 7 =:= 0.
mv(X) :- integer(X), X>=0, X mod 5 =:= 0, Y is X+1, Y mod 7 =:= 0.
```

Exemplu (tot din fișierul *testordsatconjdisj.pl* – ordinea de evaluare este importantă în recursii, iar predicatul *cut* poate fi folosit pentru separarea cazurilor în care argumentele unui predicat nu au tipul cerut de acel predicat (precum tipul *număr natural* pentru primul argument al acestui predicat pentru calculul factorialului))

```
% fact(N,F)=true <=> F = N! (N factorial)

fact(N,_) :- (not(integer(N)); N<0), write('nedefinit'), !.
fact(0,1).
fact(N,F) :- N>0, M is N-1, fact(M,G), F is G*N.

/* Daca mutam predicatul N>0 la sfarsitul acestei conjunctii si, la o interogare precum:
?- fact(5,Cat).
dam ";"/"Next" dupa raspunsul Cat=120, atunci trece la evaluarea lui fact(-1,G) si da
eroare, pentru ca, in calculul "F is G*N", argumentul G nu e instantiat. */
```

Următoarele secțiuni din acest material detaliază modul de funcționare a Prologului și **nu fac parte din materia pentru partea de programare în Prolog din examenul la Logică Matematică și Computațională**.
Vom vedea unele dintre lucrurile următoare la curs și pe altele la nivel orientativ în lecțiile de laborator.

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Ce este un predicat?

- **Predicatele** se mai numesc **propoziții cu variabile**.
- **Exemplu de predicat:** "x este un număr prim" este un predicat cu variabila x; acest predicat nu are o valoare de adevăr; pentru a obține din el un enunț cu valoare de adevăr, trebuie să–i dăm valori variabilei x.
- Variabilei x i se indică un domeniu al valorilor posibile, de exemplu $\mathbb{N}$. Se dorește ca, prin înlocuirea lui x din acest predicat cu o valoare arbitrară din acest domeniu, să se obțină o propoziție adevărată sau falsă.
- Înlocuind în acest predicat pe $x := 2 \in \mathbb{N}$, se obține propoziția adevărată "2 este un număr prim", iar înlocuind $x := 4 \in \mathbb{N}$, se obține propoziția falsă "4 este un număr prim".
- Dacă desemnăm pe $\mathbb{N}$ ca domeniu al valorilor pentru variabila x din predicatul de mai sus, atunci putem să aplicăm un cuantificator acestei variabile, obținând, astfel, tot un enunț cu valoare de adevăr: propoziția $(\forall\, x \in \mathbb{N})\,(x$ este un număr prim$)$ este falsă, în timp ce propoziția $(\exists\, x \in \mathbb{N})\,(x$ este un număr prim$)$ este adevărată. De fapt, în enunțuri cuantificate, vom considera domeniul valorilor variabilelor stabilit de la început, și nu–l vom mai preciza după variabilele cuantificate ("$\in \mathbb{N}$").

În ce fel de structuri algebrice pot lua valori variabilele?

- Așadar, pentru a exprima matematic modul de lucru cu predicate, avem nevoie nu numai de noțiunea de propoziție cu sau fără variabile și de conectori logici, ci și de un domeniu al valorilor pentru variabilele care apar în predicate (i. e. în propozițiile cu variabile).
- Pentru a descrie sistemul formal al calculului cu predicate clasic, vom avea nevoie de noțiunea de **structură de ordinul I**, reprezentând un anumit gen de structuri algebrice. Variabilele vor fi considerate ca luând valori în diverse **structuri de ordinul I**, și **clasa structurilor de ordinul I de un anumit tip** va avea asociată propria ei logică clasică cu predicate (îi vom asocia un limbaj, apoi un sistem logic bazat pe acel limbaj).
- Intuitiv, **structurile de ordinul I** sunt structuri algebrice care posedă o mulțime suport și operații, relații și constante (operații zeroare) pe această mulțime suport, i. e. operații și relații care acționează (numai) asupra elementelor mulțimii suport.
- Când, într–o structură algebrică, există operații sau relații care acționează asupra submulțimilor mulțimii suport, i. e. asupra unor mulțimi de elemente din mulțimea suport, atunci spunem că structura respectivă este o **structură de ordinul II**. În același mod (referindu–ne la mulțimi de mulțimi de elemente ș. a. m. d.) pot fi definite **structurile de ordinul III, IV etc.**.

Structuri algebrice cu operații parțiale și relații

Definiție

O *structură de ordinul I* este o structură de forma

$$\mathcal{A} = (A, (f_i)_{i \in I}, (R_j)_{j \in J}, (c_k)_{k \in K}),$$

unde:

- A este o mulțime nevidă, numită *universul structurii* $\mathcal{A}$
- I, J, K sunt mulțimi oarecare de indici (care pot fi și vide)
- pentru fiecare $i \in I$, există $n_i \in \mathbb{N}^*$, a. î. $f_i : A^{n_i} \to A$ (f_i este o operație n_i–ară pe A)
- pentru fiecare $j \in J$, există $m_j \in \mathbb{N}^*$, a. î. $R_j \subseteq A^{m_j}$ (R_j este o relație m_j–ară pe A)
- pentru fiecare $k \in K$, $c_k \in A$ (c_k este o constantă din A)

În general, operațiilor și relațiilor din componența lui $\mathcal{A}$ li se atașează indicele $\mathcal{A}$, pentru a le deosebi de simbolurile de operații și relații din limbajul pe care îl vom construi în continuare; astfel, structura de ordinul I de mai sus se notează, de regulă, sub forma: $\mathcal{A} = (A, (f_i^{\mathcal{A}})_{i \in I}, (R_j^{\mathcal{A}})_{j \in J}, (c_k^{\mathcal{A}})_{k \in K})$.

Tipuri de astfel de structuri algebrice și limbaje asociate lor

Definiție

Tipul sau *signatura* structurii de ordinul I $\mathcal{A}$ din definiția anterioară este tripletul de familii de numere naturale: $\tau := ((n_i)_{i\in I}; (m_j)_{j\in J}; (0)_{k\in K})$.
Orice structură de forma lui $\mathcal{A}$ de mai sus se numește *structură de ordinul I de tipul (sau signatura)* τ.

Exemplu

- Orice poset nevid este o structură de ordinul I de forma $\mathcal{P} = (P, \leq)$, de tipul (signatura) $\tau_1 = (\emptyset; 2; \emptyset)$ ($\leq$ este o relație binară). **Nu orice** structură de ordinul I de signatura τ_1 este un poset.
- Orice latice nevidă este o structură de ordinul I de forma $\mathcal{L} = (L, \vee, \wedge, \leq)$, de tipul (signatura) $\tau_2 = (2, 2; 2; \emptyset)$ ($\vee$ și $\wedge$ sunt operații binare (i. e. de aritate 2, i. e. cu câte două argumente), iar $\leq$ este o relație binară). **Nu orice** structură de ordinul I de signatura τ_2 este o latice.
- Orice algebră Boole este o structură de ordinul I de forma $\mathcal{B} = (B, \vee, \wedge, \bar{\ }, \leq, 0, 1)$, de tipul (signatura) $\tau_3 = (2, 2, 1; 2; 0, 0)$ ($\vee$ și $\wedge$ sunt operații binare, $\bar{\ }$ este o operație unară, $\leq$ este o relație binară, iar 0 și 1 sunt constante (operații zeroare, i. e. de aritate zero, i. e. fără argumente)). **Nu orice** structură de ordinul I de signatura τ_3 este o algebră Boole.

Fiecărei signaturi τ a unei structuri de ordinul I (fiecărei clase de structuri de ordinul I de o anumită signatură τ) i se asociază un limbaj, numit **limbaj de ordinul I** și notat, de obicei, cu $\mathcal{L}_\tau$, în care pot fi exprimate proprietățile algebrice ale structurilor de ordinul I de signatura τ.

Pentru cele ce urmează **fixăm** o **signatură** arbitrară $\tau := ((n_i)_{i\in I}; (m_j)_{j\in J}; (0)_{k\in K})$.

- **Alfabetul** limbajului de ordinul I $\mathcal{L}_\tau$ este format din următoarele **simboluri primitive**:
  1. o mulțime infinită de **variabile**: $Var = \{x, y, z, u, v, \ldots\}$;
  2. **simboluri de operații**: $(f_i)_{i\in I}$; pentru fiecare $i \in I$, numărul natural nenul n_i se numește *ordinul* sau *aritatea* lui f_i;
  3. **simboluri de relații**: $(R_j)_{j\in J}$; pentru fiecare $j \in J$, numărul natural nenul m_j se numește *ordinul* sau *aritatea* lui R_j;
  4. **simboluri de constante**: $(c_k)_{k\in K}$;
  5. **simbolul de egalitate**: $=$ (un semn egal îngroșat) (*a nu se confunda cu egalul simplu!*);
  6. **conectorii logici primitivi**: $\neg$ (*negația*), $\to$ (*implicația*);
  7. **cuantificatorul universal**: $\forall$ (*oricare ar fi*)
  8. paranteze: $(,), [,]$, precum și virgula.
- Pentru comoditate, vom spune uneori: "operații", "relații" și "constante" în loc de "simboluri de operații/relații/constante", respectiv.

Mulțimile de simboluri de la punctele ①–⑧ se consideră două câte două disjuncte.

Elementele lui *Var* nu sunt variabile propoziționale, ci sunt variabilele care apar în predicate, i.e. în propozițiile cu variabile.
$\circledast$ Mulți autori consideră virgula ca având semnificație implicită, subînțeleasă, în scrierea termenilor și a formulelor atomice, și nu includ virgula în limbajul $\mathcal{L}_{\mathcal{T}}$.
$\circledast$ La fel putem considera parantezele care încadrează argumentele unui termen sau ale unei formule atomice – a se vedea mai jos.
Dar nu sunt obligatorii distincțiile $\circledast$ și $\circledast$ între simbolurile din acest alfabet.
Iar parantezele din cadrul formulelor (putem include aici toate parantezele, precum și virgula) care stabilesc ordinea aplicării conectorilor logici și a cuantificatorilor pot fi tratate ca în logica propozițională clasică: formulele cu același arbore asociat se consideră a fi egale, indiferent de parantezările acestora – de data aceasta, arborii vor fi oarecare, nu neapărat binari; a se vedea mai jos.

Termenii și formulele atomice

Definiție

Termenii limbajului $\mathcal{L}_\tau$ se definesc, recursiv, astfel:

1. variabilele și simbolurile de constante sunt termeni;
2. dacă $n \in \mathbb{N}^*$, f este un simbol de operație n–ară și $t_1, \ldots, t_n$ sunt termeni, atunci $f(t_1, \ldots, t_n)$ este un termen;
3. orice termen se obține prin aplicarea regulilor ① și ② de un număr finit de ori.

f se numește *operația dominantă* sau *operatorul dominant* al termenului $f(t_1, \ldots, t_n)$.

Următoarea definiție **nu** este recursivă:

Definiție (**formule atomice**: relații aplicate unor termeni)

Formulele atomice ale limbajului $\mathcal{L}_\tau$ se definesc astfel:

1. dacă t_1 și t_2 sunt termeni, atunci $t_1 = t_2$ este o formulă atomică;
2. dacă $m \in \mathbb{N}^*$, R este un simbol de relație m–ară și $t_1, \ldots, t_m$ sunt termeni, atunci $R(t_1, \ldots, t_m)$ este o formulă atomică;
3. orice formulă atomică este de forma de la ① sau de forma de la ②.

Formulele

Definiție (obț. din formule atomice aplicând conectori și cuantificatori)

Formulele limbajului $\mathcal{L}_\tau$ se definesc, recursiv, astfel:

1. formulele atomice sunt formule;
2. dacă φ este o formulă, atunci $\neg\varphi$ este o formulă;
3. dacă φ și ψ sunt formule, atunci $\varphi \to \psi$ este o formulă;
4. dacă φ este o formulă și x este o variabilă, atunci $\forall x\varphi$ este o formulă;
5. orice formulă se obține prin aplicarea regulilor ①, ②, ③ și ④ de un număr finit de ori.

Definiție (cele obținute din formule atomice aplicând conectori logici)

Formulele fără cuantificatori ale limbajului $\mathcal{L}_\tau$ se definesc, recursiv, astfel:

1. formulele atomice sunt formule fără cuantificatori;
2. dacă φ este o formulă fără cuantificatori, atunci $\neg\varphi$ este o formulă fără cuantificatori;
3. dacă φ și ψ sunt formule fără cuantificatori, atunci $\varphi \to \psi$ este o formulă fără cuantificatori;
4. orice formulă fără cuantificatori se obține prin aplicarea regulilor ①, ② și ③ de un număr finit de ori.

Notație

- $Term(\mathcal{L}_\tau) :=$ mulțimea termenilor limbajului $\mathcal{L}_\tau$ (notație ad–hoc);
- $FormAt(\mathcal{L}_\tau) :=$ mulțimea formulelor atomice limbajului $\mathcal{L}_\tau$ (notație ad–hoc);
- $Form(\mathcal{L}_\tau) :=$ mulțimea formulelor limbajului $\mathcal{L}_\tau$.

Remarcă

Conform definițiilor anterioare:

- $Term(\mathcal{L}_\tau)$ este cea mai mică mulțime de cuvinte finite și nevide peste alfabetul de mai sus care include mulțimea *Var* a variabilelor și mulțimea simbolurilor de constante și este închisă la simbolurile de operații (cu argumente, i.e. de aritate nenulă);
- $Form(\mathcal{L}_\tau)$ este cea mai mică mulțime de cuvinte finite și nevide peste alfabetul de mai sus care include mulțimea $FormAt(\mathcal{L}_\tau)$ a formulelor atomice și este închisă la aplicările conectorilor logici $\neg$ și $\rightarrow$ și a cuantificatorului $\forall$;
- analog pentru mulțimea formulelor fără cuantificatori.

Finitudinea acestor cuvinte este asigurată de regula ③ din definiția termenilor, definiția formulelor atomice și regula ⑤ din definiția formulelor. Termenii, formulele și formulele fără cuantificatori pot fi definite și ca fiind cuvinte (apriori presupuse) finite care se obțin prin aplicări succesive ale regulilor din definițiile lor, ceea ce va implica faptul că aplicările acelor reguli se vor face de un număr finit de ori, ca în cazul enunțurilor din logica propozițională clasică.

Arborii oarecare asociați termenilor și formulelor

Fie $v \in Var$, c un simbol de constantă, $n \in \mathbb{N}^*$, $t_1, \ldots, t_n$ termeni, f un simbol de operație n–ară, R un simbol de operație n–ară, $\varphi, \psi \in Form(\mathcal{L}_\tau)$, arbitrare.
Arborii asociați termenilor, formulelor atomice, respectiv formulelor se definesc, recursiv, astfel:

Notație

Introducem abrevierile: pentru orice formule φ, ψ și orice variabilă x:

- **conectorii logici derivați** $\vee$ (*disjuncția*), $\wedge$ (*conjuncția*) și $\leftrightarrow$ (*echivalența*) se definesc astfel:
  $\varphi \vee \psi := \neg\varphi \to \psi$
  $\varphi \wedge \psi := \neg(\varphi \to \neg\psi)$
  $\varphi \leftrightarrow \psi := (\varphi \to \psi) \wedge (\psi \to \varphi)$
- **cuantificatorul existențial** $\exists$ (*există*) se definește astfel:
  $\exists x\varphi := \neg\forall x\neg\varphi$

(Convenție privind scrierea conectorilor logici, a cuantificatorilor și a simbolului de egalitate, pentru evitarea excesului de paranteze)

- $\neg, \forall, \exists$ vor avea prioritate mai mare;
- $\to, \vee, \wedge, \leftrightarrow, =$ vor avea prioritate mai mică.

Exemplu

Considerăm signatura $(2, 1; 2, 1; 0)$ și limbajul de ordinul I de această signatură conținând simbolul de operație binară f, simbolul de operație unară g, simbolul de relație binară R, simbolul de relație unară S și simbolul de constantă c (astfel că o algebră de această signatură este de forma $\mathcal{A} = (A, f^{\mathcal{A}}, g^{\mathcal{A}}, R^{\mathcal{A}}, S^{\mathcal{A}}, c^{\mathcal{A}})$, cu: $f^{\mathcal{A}} : A^2 \to A$, $g^{\mathcal{A}} : A \to A$, $R^{\mathcal{A}} \subseteq A^2$, $S^{\mathcal{A}} \subseteq A$ și $c^{\mathcal{A}} \in A$). Fie $x, y \in Var$ și:

$$\varphi = \exists x \forall y\, [f(x, g(y)) = f(f(x, y), c) \wedge (R(g(x), f(g(x), c)) \vee S(g(g(c))))].$$

Vom vedea că φ este un tip de formulă numit **enunț**. φ este o abreviere pentru:

$$\varphi = \exists x \forall y\, [f(x, g(y)) = f(f(x, y), c) \wedge (\neg R(g(x), f(g(x), c)) \rightarrow S(g(g(c))))] =$$

$$\exists x \forall y \neg [f(x, g(y)) = f(f(x, y), c) \rightarrow \neg (\neg R(g(x), f(g(x), c)) \rightarrow S(g(g(c))))] =$$

$$\neg \forall x \neg \forall y \neg [f(x, g(y)) = f(f(x, y), c) \rightarrow \neg (\neg R(g(x), f(g(x), c)) \rightarrow S(g(g(c))))],$$

așadar φ are următorul arbore binar asociat:

Notație (mulțimile din această notație vor fi definite recursiv mai jos)

Pentru orice termen t și orice formulă φ, notăm:

- $V(t) :=$ mulțimea variabilelor termenului t
- $V(\varphi) :=$ mulțimea variabilelor formulei φ
- $FV(\varphi) :=$ mulțimea variabilelor *libere* ale formulei φ

Definiție (variabilele dintr–un termen)

Pentru orice termen t:

- dacă $t = x$, unde x este o variabilă, atunci $V(t) = V(x) := \{x\}$
- dacă $t = c$, unde c este o constantă, atunci $V(t) = V(c) := \emptyset$
- dacă $t = f(t_1, \ldots, t_n)$, unde $n \in \mathbb{N}^*$, f este un simbol de operație n–ară și $t_1, \ldots, t_n$ sunt termeni, atunci $V(t) := \bigcup_{i=1}^{n} V(t_i)$

Definiție (variabilele și variabilele libere dintr–o formulă)

Pentru orice formulă φ:

- dacă $\varphi = (t_1 = t_2)$, unde t_1 și t_2 sunt termeni, atunci $V(\varphi) = FV(\varphi) := V(t_1) \cup V(t_2)$

Definiție (variabilele și variabilele libere dintr–o formulă – continuare)

- dacă $\varphi = R(t_1, \ldots, t_m)$, unde $m \in \mathbb{N}^*$, R este un simbol de relație m–ară și $t_1, \ldots, t_m$ sunt termeni, atunci $V(\varphi) = FV(\varphi) := \bigcup_{i=1}^{m} V(t_i)$
- dacă $\varphi = \neg\psi$, pentru o formulă ψ, atunci $V(\varphi) := V(\psi)$ și $FV(\varphi) := FV(\psi)$
- dacă $\varphi = \psi \to \chi$, pentru două formule ψ, χ, atunci $V(\varphi) := V(\psi) \cup V(\chi)$ și $FV(\varphi) := FV(\psi) \cup FV(\chi)$
- dacă $\varphi = \forall x\psi$, pentru o formulă ψ și o variabilă x, atunci $V(\varphi) := V(\psi) \cup \{x\}$ și $FV(\varphi) := FV(\psi) \setminus \{x\}$

Remarcă

Este imediat, din definiția anterioară și definiția conectorilor logici derivați și a cuantificatorului existențial, că, pentru orice formule ψ, χ și orice variabilă x:

- $V(\psi \vee \chi) = V(\psi \wedge \chi) = V(\psi \leftrightarrow \chi) = V(\psi) \cup V(\chi)$ și $FV(\psi \vee \chi) = FV(\psi \wedge \chi) = FV(\psi \leftrightarrow \chi) = FV(\psi) \cup FV(\chi)$
- $V(\exists x\psi) = V(\psi) \cup \{x\}$ și $FV(\exists x\psi) = FV(\psi) \setminus \{x\}$

Enunțurile sunt formulele fără variabile libere, deci cu toate variabilele **legate**:

Definiție (variabile libere și variabile legate)

Pentru orice variabilă x și orice formulă φ:

- dacă $x \in FV(\varphi)$, atunci x se numește *variabilă liberă a lui* φ;
- dacă $x \in V(\varphi) \setminus FV(\varphi)$ (prin extensie, chiar dacă $x \in Var \setminus FV(\varphi)$), atunci x se numește *variabilă legată a lui* φ;
- dacă $FV(\varphi) = \emptyset$ (i. e. φ nu are variabile libere), atunci φ se numește *enunț*.

Exemplu

- În formula $\exists x(x^2{=}x)$, x este variabilă legată. Această formulă este un enunț.
- În formula $\forall y \forall z(z \cdot x \leq y \cdot z)$, x este variabilă liberă.

Observație (diferența dintre tipurile de variabile, intuitiv)

- **Variabilele libere** sunt variabilele care nu intră sub incidența unui cuantificator, variabilele cărora "avem libertatea de a le da valori".
- **Variabilele legate** sunt variabilele care intră sub incidența unui cuantificator, deci sunt destinate parcurgerii unei întregi mulțimi de valori.

Remarcă

Formulele fără cuantificatori sunt exact formulele φ cu $V(\varphi) = FV(\varphi)$, adică exact formulele fără variabile legate, i.e. formulele în care toate variabilele sunt libere (adică nu sunt cuantificate, nu li se aplică niciun cuantificator).

Notație (cum specificăm că nu avem alte variabile (libere))

Fie $n \in \mathbb{N}^*$ și $x_1, \ldots, x_n$ variabile.
Dacă t este un termen cu $V(t) \subseteq \{x_1, \ldots, x_n\}$, atunci vom nota $t(x_1, \ldots, x_n)$.
Dacă φ este o formulă cu $FV(\varphi) \subseteq \{x_1, \ldots, x_n\}$, atunci vom nota $\varphi(x_1, \ldots, x_n)$.

Definiție (vom vedea că $\varphi(t, x_1, \ldots, x_n) = \sigma(\varphi)$, unde σ este substituția: $\sigma : Var \to Term(\mathcal{L}_\tau)$, $\sigma(v) = \begin{cases} t, & v = x, \\ v, & v \in Var \setminus \{x\} \end{cases}$)

Fie $n \in \mathbb{N}$, $\varphi(x, x_1, \ldots, x_n) \in Form(\mathcal{L}_\tau)$ cu $FV(\varphi) = \{x, x_1, \ldots, x_n\} \subset Var$ și $t \in Term(\mathcal{L}_\tau)$. Formula obținută din φ prin *substituția lui x cu t* se notează cu $\varphi(t, x_1, \ldots, x_n)$ și se definește astfel:

- fiecare $y \in (V(\varphi) \setminus FV(\varphi)) \cap V(t) = (V(\varphi) \setminus \{x, x_1, \ldots, x_n\}) \cap V(t)$ (variabilă legată a lui φ care apare și în t) se înlocuiește cu o variabilă $v \notin V(t) \cup V(\varphi)$;
- apoi se înlocuiește x cu t.

Exemplu

Fie variabilele x, y, z, formula $\varphi(x) := \exists\, y(x < y)$ și termenul $t := y + z$.
Atunci $\varphi(t)$ se obține astfel:

- $\varphi(x) = \exists\, y(x < y)$ se înlocuiește cu $\exists\, v(x < v)$;
- prin urmare, $\varphi(t) = \exists\, v(y + z < v)$.

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Vom vedea că un program în Prolog definește o întreagă clasă de structuri de ordinul I

$femeie(ana).$ $femeie(carmen).$ $femeie(elena).$ $femeie(eva).$
$femeie(maria).$ $femeie(sara).$ $femeie(valentina).$
$barbat(adam).$ $barbat(constantin).$ $barbat(eugen).$ $barbat(george).$
$barbat(ion).$ $barbat(iosif).$ $barbat(tudor).$ $barbat(victor).$
$parinte(sara, adam).$ $parinte(sara, eva).$
$parinte(constantin, iosif).$ $parinte(constantin, elena).$
$parinte(valentina, ion).$ $parinte(valentina, sara).$
$parinte(victor, constantin).$ $parinte(victor, maria).$
$parinte(ana, victor).$ $parinte(ana, valentina).$
$parinte(carmen, victor).$ $parinte(carmen, valentina).$
$parinte(george, victor).$ $parinte(george, valentina).$
$parinte(eugen, tudor).$ $parinte(eugen, ana).$
$frati(X, Y)$:- $X \backslash= Y$, $parinte(X, P)$, $parinte(Y, P)$.
$tata(X, T)$:- $parinte(X, T)$, $barbat(T)$.
$mama(X, M)$:- $parinte(X, M)$, $femeie(M)$.
$unchi(X, U)$:- $parinte(X, P)$, $frati(P, U)$, $barbat(U)$.
$matusa(X, M)$:- $parinte(X, P)$, $frati(P, M)$, $femeie(M)$.
$bunic(X, B)$:- $parinte(X, P)$, $parinte(P, B)$.
$stramos(X, X, 0)$. % al treilea argument e numarul de generatii
$stramos(X, S, G)$:- $parinte(X, P)$, $stramos(P, S, H)$, G is $H + 1$.

În programul în Prolog de mai sus:

- X, P, T, M, U, B, S, G, H sunt **variabile**;
- $+$ este **operație**;
- *femeie*, *barbat*, *frati*, *parinte*, *tata*, *mama*, *unchi*, *matusa*, *bunic*, *stramos* sunt **predicate**;
- $H + 1$ este un **termen**;
- *ana*, *carmen*, *elena*, *eva*, *maria*, *sara*, *valentina*, *adam*, *constantin*, *eugen*, *george*, *ion*, *iosif*, *tudor*, *victor*, 0 și 1 sunt **constante** Prolog;
- de exemplu, $femeie(eva)$, $frati(victor, george)$, $parinte(victor, ion)$, $parinte(X, M)$, $stramos(X, X, 0)$, $stramos(X, S, G)$ sunt **formule atomice**.

REGULILE și ultimul FAPT din programul anterior corespund **formulelor**:

(atenție: toate variabilele cuantificate UNIVERSAL)

- $(\forall X)(\forall Y)(\forall P)[(\neg(X = Y) \wedge parinte(X, P) \wedge parinte(Y, P)) \rightarrow frati(X, Y)]$
- $(\forall X)(\forall T)[(parinte(X, T) \wedge barbat(T)) \rightarrow tata(X, T)]$
- $(\forall X)(\forall M)[(parinte(X, M) \wedge femeie(M)) \rightarrow mama(X, M)]$
- $(\forall X)(\forall U)(\forall P)[(parinte(X, P) \wedge frati(P, U) \wedge barbat(U)) \rightarrow unchi(X, U)]$
- $(\forall X)(\forall M)(\forall P)[(parinte(X, P) \wedge frati(P, M) \wedge femeie(M)) \rightarrow matusa(X, M)]$
- $(\forall X)(\forall B)[(parinte(X, P) \wedge parinte(P, B)) \rightarrow bunic(X, B)]$

- $(\forall X)\,(stramos(X, X, 0))$
- $(\forall X)\,(\forall S)\,(\forall G)\,[(parinte(X, P) \wedge stramos(P, S, H) \wedge (G = H + 1)) \rightarrow stramos(X, S, G)]$

INTEROGĂRILE:

?- *parinte*(*C*, *victor*). % Care sunt copiii lui Victor?
?- *parinte*(*F*, *victor*), *femeie*(*F*). % Care sunt fiicele lui Victor?
?- *stramos*(*carmen*, *S*, 3), *barbat*(*S*). /* Care sunt strabunicii (cunoscuti, adica din baza de cunostinte de mai sus, ai) lui Carmen? */
?- *tata*(*X*, *T*), *stramos*(*T*, *elena*, *G*). /* Pentru cine este Elena stramos din partea tatalui, cine este tatal acelui urmas si cate generatii sunt intre tata si Elena? */

corespund **formulelor**:

(atenție: toate variabilele cuantificate EXISTENŢIAL)

- $(\exists C)\,(parinte(C, victor))$
- $(\exists F)\,(parinte(F, victor) \wedge femeie(F))$
- $(\exists S)\,(stramos(carmen, S, 3) \wedge barbat(S))$
- $(\exists X)\,(\exists T)\,(\exists G)\,(tata(X, T) \wedge stramos(T, elena, G))$

Care este **signatura** limbajului de ordinul I definit în programul Prolog de mai sus?

Pentru început, să considerăm programul anterior, mai puțin definiția predicatului **stramos**:

$femeie(ana).$ $femeie(carmen).$ $femeie(elena).$ $femeie(eva).$
$femeie(maria).$ $femeie(sara).$ $femeie(valentina).$
$barbat(adam).$ $barbat(constantin).$ $barbat(eugen).$ $barbat(george).$
$barbat(ion).$ $barbat(iosif).$ $barbat(tudor).$ $barbat(victor).$
$parinte(sara, adam).$ $parinte(sara, eva).$
$parinte(constantin, iosif).$ $parinte(constantin, elena).$
$parinte(valentina, ion).$ $parinte(valentina, sara).$
$parinte(victor, constantin).$ $parinte(victor, maria).$
$parinte(ana, victor).$ $parinte(ana, valentina).$
$parinte(carmen, victor).$ $parinte(carmen, valentina).$
$parinte(george, victor).$ $parinte(george, valentina).$
$parinte(eugen, tudor).$ $parinte(eugen, ana).$
$frati(X, Y)$:- $X \backslash= Y$, $parinte(X, P)$, $parinte(Y, P)$.
$tata(X, T)$:- $parinte(X, T)$, $barbat(T)$.
$mama(X, M)$:- $parinte(X, M)$, $femeie(M)$.
$unchi(X, U)$:- $parinte(X, P)$, $frati(P, U)$, $barbat(U)$.
$matusa(X, M)$:- $parinte(X, P)$, $frati(P, M)$, $femeie(M)$.
$bunic(X, B)$:- $parinte(X, P)$, $parinte(P, B)$.

Signatura corespunzătoare programului de mai sus este

$$\tau = (\underbrace{\emptyset}_{\text{aritățile operațiilor}}; \underbrace{2,2,2,2,2,2,2,1,1}_{\text{aritățile relațiilor}}; \underbrace{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}_{\text{aritățile constantelor}}).$$

Amintesc că *aritățile* sunt numerele de argumente.

Structura de ordinul I de această signatură definită în programul de mai sus este:
$\mathcal{A} = (A; \emptyset; \mathit{frati}, \mathit{parinte}, \mathit{tata}, \mathit{mama}, \mathit{unchi}, \mathit{matusa}, \mathit{bunic}, \mathit{femeie}, \mathit{barbat};$
$\mathit{ana}, \mathit{carmen}, \mathit{elena}, \mathit{eva}, \mathit{maria}, \mathit{sara}, \mathit{valentina}, \mathit{adam}, \mathit{constantin}, \mathit{eugen}, \mathit{george}, \mathit{ion},$
$\mathit{iosif}, \mathit{tudor}, \mathit{victor})$, cu:

- mulțimea suport (i.e. mulțimea elementelor) $A = \{\mathit{ana}, \mathit{carmen}, \mathit{elena}, \mathit{eva}, \mathit{maria}, \mathit{sara}, \mathit{valentina}, \mathit{adam}, \mathit{constantin}, \mathit{eugen}, \mathit{george}, \mathit{ion}, \mathit{iosif}, \mathit{tudor}, \mathit{victor}\}$;
- nicio operație;
- 7 predicate (i.e. relații) binare (i.e. de aritate 2, i.e. cu 2 argumente): $\mathit{frati}, \mathit{parinte}, \mathit{tata}, \mathit{mama}, \mathit{unchi}, \mathit{matusa}, \mathit{bunic} \subseteq A^2$;
- 2 predicate (i.e. relații) unare (i.e. de aritate 1, i.e. cu 1 argument): $\mathit{femeie}, \mathit{barbat} \subseteq A$;
- 15 constante (amintesc că acestea sunt operațiile zeroare, operațiile de aritate 0, i.e. operațiile fără argumente): $\mathit{ana}, \mathit{carmen}, \mathit{elena}, \mathit{eva}, \mathit{maria}, \mathit{sara}, \mathit{valentina}, \mathit{adam}, \mathit{constantin}, \mathit{eugen}, \mathit{george}, \mathit{ion}, \mathit{iosif}, \mathit{tudor}, \mathit{victor} \in A$.

Primele fapte din acest program definesc relațiile unare *femeie* și *barbat*:

- $femeie = \{ana, carmen, elena, eva, maria, sara, valentina\}$;
- $barbat = \{adam, constantin, eugen, george, ion, iosif, tudor, victor\}$.

Pentru orice $x \in A$, faptul că $x \in femeie$ se mai scrie: $femeie(x)$. La fel pentru relația unară *barbat*.

Următoarele fapte definesc relația binară *parinte*:

- $parinte = \{(sara, adam), (sara, eva), (constantin, iosif), (constantin, elena), (valentina, ion), (valentina, sara), (victor, constantin), (victor, maria), (ana, victor), (ana, valentina), (carmen, victor), (carmen, valentina), (george, victor), (george, valentina), (eugen, tudor), (eugen, ana)\}$.

Pentru orice $x, y \in A$, faptul că $(x, y) \in parinte$ se mai scrie: $parinte(x, y)$. La fel pentru următoarele relații binare.

Regulile următoare definesc relațiile binare *frati*, *tata*, *mama*, *unchi* și *matusa*:

- $frati = \{(x, y) \in A^2 \mid x \neq y, (\exists\, p \in A)\,((x, p), (y, p) \in parinte)\}$;
- $tata = \{(x, t) \in A^2 \mid (x, t) \in parinte, t \in barbat\}$;
- $mama = \{(x, m) \in A^2 \mid (x, m) \in parinte, m \in femeie\}$;
- $unchi = \{(x, u) \in A^2 \mid (\exists\, p \in A)\,((x, p) \in parinte, (p, u) \in frati, u \in barbat)\}$;
- $matusa = \{(x, m) \in A^2 \mid (\exists\, p \in A)\,((x, p) \in parinte, (p, m) \in frati, m \in femeie)\}$;
- $bunic = \{(x, b) \in A^2 \mid (\exists\, p \in A)\,((x, p), (p, b) \in parinte)\}$.

Și acum să reanalizăm programul inițial, care include și predicatul ternar (relația ternară, i.e. de aritate 3, i.e. cu 3 argumente) *stramos* $\subseteq A \times A \times \mathbb{N}$, definită prin ultimul fapt și ultima regulă, care dau următoarea definiție recursivă:

- $\{(x, x, 0) \mid x \in A\} \subseteq$ *stramos* și:
- pentru orice $x, s, p \in A$ și orice $h \in \mathbb{N}$, dacă $(x, p) \in$ *parinte* și $(p, s, h) \in$ *stramos*, atunci $(x, s, h + 1) \in$ *stramos*.

Pentru orice $x, s \in A$ și orice $g \in \mathbb{N}$, faptul că $(x, s, g) \in$ *stramos* se mai scrie *stramos*(x, s, g).

Dar *stramos* nu este o relație ternară pe A, ci are primele două argumente din A, iar al treilea din $\mathbb{N}$!
În plus, 0 și 1 sunt constante din $\mathbb{N}$, nu din A, iar $+ : \mathbb{N}^2 \to \mathbb{N}$, deci $+$ este o operație binară pe $\mathbb{N}$, nu pe A.

Așadar cadrul de lucru din secțiunea anterioară a acestui curs trebuie extins!
Putem face acest lucru înlocuind mulțimea suport a structurii $\mathcal{A}$ cu $A \cup \mathbb{N}$ și permițând operațiilor să fie parțial definite.

Definiție (mnemonic din cursul de logică matematică)

Dacă M și N sunt mulțimi, atunci o *funcție parțială* (*relație binară funcțională*) de la M la N este o relație binară între M și N $\varphi \subseteq M \times N$ cu proprietatea că:

pentru orice $x \in M$, există **cel mult un** $y \in N$ cu proprietatea că $(x, y) \in \varphi$.

Definiție (continuare)

Dacă, pentru un $x \in M$, există un $y \in N$ cu $(x, y) \in \varphi$, atunci, ca în cazul *funcțiilor* (i.e. al *relațiilor binare funcționale totale*), se notează $\varphi(x) = y$ și acest $y \in N$ se numește *valoarea funcției parțiale* φ *în punctul* x.

Notație

Dacă M și N sunt mulțimi, atunci faptul că φ este o funcție parțială de la M la N se notează:

$$\varphi : M \multimap N.$$

Exemplu (să analizăm această soluție pe un tip cunoscut de algebră)

Dacă $\mathcal{V}$ este un spațiu vectorial, format din:

- un grup abelian $(G, \circ, ^{-}, e)$, cu $\circ : G^2 \to G$ (operație binară pe G: legea de compoziție), $^{-} : G \to G$ (operație unară pe G: inversarea) și $e \in G$ (constantă din G, i.e. operație zeroară pe G: elementul neutru),
- un corp $(K, +, \cdot, -, ^{-1}, 0, 1)$, cu $+ : K^2 \to K$ și $\cdot : K^2 \to K$ (operații binare pe K), $- : K \to K$ și $^{-1} : K \to K$ (operații unare pe K), iar $0, 1 \in K$ (constante din K, i.e. operații zeroare pe K),
- și o lege de compoziție între scalarii din K și vectorii din G: $* : K \times G \to G$.

O structură algebrică având mulțimea suport $K \times V$ ar trebui să aibă orice operație f de aritate $n \in \mathbb{N}$ de forma $f : (K \times V)^n \to K \times V$, iar, în cazul unei

Exemplu (continuare)

structuri cu mulțimea suport $K \cup V$, o operație g de aritate n ar trebui să fie o funcție $g : (K \cup V)^n \to K \cup V$.
Așadar, putem privi spațiul vectorial $\mathcal{V}$ ca pe o structură algebrică având mulțimea suport $K \cup V$ dacă permitem operațiilor sale să fie funcții parțiale.
O altă posibilitate este să privim spațiul vectorial $\mathcal{V}$ ca pe o structură algebrică având două mulțimi suport: K și G, mai precis familia de mulțimi (K, G), cu operațiile total definite, dar astfel încât, pentru orice $m, p \in \mathbb{N}$, o operație h de aritate $m + p$ (i.e. cu $m + p$ argumente) a lui $\mathcal{V}$ să poată fi de forma:
$h : K^m \times G^p \to K$ sau $h : K^m \times G^p \to G$.
O astfel de structură algebrică se numește *algebră bisortată*, i.e. *cu două sorturi*, cu două mulțimi suport, două tipuri de elemente, în cazul acesta: SCALARI din K și VECTORI din G.

La fel ca în exemplul anterior, o *algebră multisortată*, i.e. *cu mai multe sorturi*, cu mai multe mulțimi suport, mai multe tipuri de elemente, și care, pe lângă operații, e înzestrată și cu, relații, va fi o algebră de forma:
$\mathcal{M} = ((M_s)_{s \in S}; (f_i)_{i \in I}; (\rho_j)_{j \in J}; (c_k)_{k \in K})$, unde S e *mulțimea de sorturi*, elementele structurii algebrice $\mathcal{M}$ sunt elementele mulțimilor din familia de mulțimi $(M_s)_{s \in S}$, $(f_i)_{i \in I}$ este familia de operații, $(\rho_j)_{j \in J}$ este familia de relații, iar $(c_k)_{k \in K}$ este familia de constante ale structurii algebrice $\mathcal{M}$:

Pentru orice mulțime **nevidă** S, o *algebră S–sortată* înzestrată și cu operații, și cu, relații este o structură algebrică $\mathcal{M} = ((M_s)_{s \in S}; (f_i)_{i \in I}; (\rho_j)_{j \in J}; (c_k)_{k \in K})$, unde:

- elementele lui S se numesc *sorturi* (intuitiv, *sorturile* sunt indici corespunzători TIPURILOR DE ELEMENTE, în acest caz TIPURILOR DE DATE cu care lucrează programul de mai sus);
- $(M_s)_{s \in S}$ este *mulțimea S–sortată* a **elementelor** lui $\mathcal{M}$, adică M este o familie de mulțimi indexată de S și, pentru fiecare $s \in S$, elementele lui M_s se numesc *elementele lui $\mathcal{M}$ de sort s*;
- I, J, K sunt mulțimi, nu neapărat nevide;
- pentru fiecare $i \in I$, există $n_i \in \mathbb{N}^*$ și sorturile $s_{i,1}, \ldots, s_{i,n_i}, s_i \in S$ astfel încât $f_i : M_{s_{i,1}} \times \ldots \times M_{s_{i,n_i}} \to M_{s_i}$: f_i este o operație de aritate n_i a lui $\mathcal{M}$;
- pentru fiecare $j \in J$, există $m_j \in \mathbb{N}^*$ și sorturile $t_{j,1}, \ldots, t_{j,m_j} \in S$ astfel încât $\rho_j \subseteq M_{t_{j,1}} \times \ldots \times M_{t_{j,m_j}} \to M_{t_j}$: ρ_j este o relație de aritate m_j a lui $\mathcal{M}$;
- pentru fiecare $k \in K$, există sortul $u_k \in S$ astfel încât $c_k \in M_{u_k}$: c_k este o constantă a lui $\mathcal{M}$.

Putem defini structura de ordinul I corespunzătoare programului de mai sus ca pe o algebră bisortată, cu mulțimea suport $(A, \mathbb{N})$.
Desigur, operația binară $+$ pe $\mathbb{N}$ nu este definită în programul de mai sus, ci este predefinită în Prolog, împreună cu alte operații aritmetice și relații, de exemplu $<$, $=<$ etc..

Să mai observăm că relațiile (predicatele) pot fi privite ca operații având *sortul rezultatului* BOOLEAN: adăugăm o a treia mulțime suport, conținând valorile booleene: $\{\text{false}, \text{true}\}$ sau $\mathcal{L}_2=\{0,1\}$ ($0 = \text{false}$, $1 = \text{true}$), astfel că structura algebrică asociată programului de mai sus devine o *algebră trisortată*, cu mulțimea suport $(A_s)_{s\in\overline{1,3}} = (A, \mathbb{N}, \{\text{false}, \text{true}\})$, și orice relație din această algebră devine o operație cu valori în $\{\text{false}, \text{true}\}$: pentru orice $n \in \mathbb{N}^*$ și orice relație n–ară $\rho \subseteq A_{s_1} \times \ldots \times A_{s_n}$, unde $s_1, \ldots, s_n$ sunt SORTURI, în acest caz aparținând mulțimii de sorturi $\overline{1,3}$, în Prolog această relație este implementată ca o operație

$$\rho' : A_{s_1} \times \ldots \times A_{s_n} \to \{\text{false}, \text{true}\},$$

definită astfel: pentru orice $x_1 \in A_{s_1}, \ldots, x_n \in A_{s_n}$,

$$\rho'(x_1, \ldots, x_n) = \begin{cases} \text{true}, & \text{dacă } (x_1, \ldots, x_n) \in \rho, \\ \text{false}, & \text{altfel.} \end{cases}$$

De acum încolo, operația ρ' va fi notată tot ρ. De exemplu, pentru programul de mai sus:
$frati(ana, carmen) = \text{true}$,
$frati(ana, eva) = \text{false}$,
pentru orice $X \in A$, $stramos(X, X, 0) = \text{true}$, iar $stramos(X, X, 1) = \text{false}$.
Iar, cum **constantele** sunt *operații zeroare (nulare)*, i.e. operații de aritate 0, operații fără argumente, nu e nevoie să facem distincție între constante și operații.

Ce este o *algebră multisortată* înzestrată numai cu OPERAŢII? Aceasta este noțiunea propriu–zisă de algebră multisortată.

Definiție

Pentru orice mulțime **nevidă** S, o *algebră S–sortată* este o structură algebrică $\mathcal{M} = ((M_s)_{s \in S}; (f_i)_{i \in I})$, unde:

- elementele lui S se numesc *sorturi* (repet că *sorturile* sunt indici corespunzători TIPURILOR DE ELEMENTE, TIPURILOR DE DATE);
- mulțimea suport, i.e. mulțimea *elementelor* algebrei $\mathcal{M}$ este *mulțimea S–sortată*, adică familia de mulțimi indexată de S $(M_s)_{s \in S}$; pentru fiecare $s \in S$, elementele lui M_s se numesc *elementele lui $\mathcal{M}$ de sort s*;
- $(f_i)_{i \in I}$ este familia *operațiilor* lui $\mathcal{M}$; pentru fiecare $i \in I$, există $n_i \in \mathbb{N}$ și sorturile $s_{i,1}, \ldots, s_{i,n_i}, s_i \in S$ astfel încât $f_i : M_{s_{i,1}} \times \ldots \times M_{s_{i,n_i}} \to M_{s_i}$: f_i este o operație de aritate n_i a lui $\mathcal{M}$; dacă $n_i = 0$, atunci această declarație a lui f_i devine: $f_i \in M_{s_i}$ (a se revedea discuția dintr–un curs anterior privind operațiile zeroare).

Așadar ce structură algebrică multisortată corespunde programului anterior în Prolog? Cum am procedat și mai sus, să scriem operațiile (acestea incluzând relațiile, predicatele) descrescător după arități.

(Algebra trisortată definită în programul în Prolog de mai sus)

$\mathcal{A} = ((A, \mathbb{N}, \{\text{false}, \text{true}\}); \mathit{stramos}, +, \mathit{frati}, \mathit{parinte}, \mathit{tata}, \mathit{mama}, \mathit{unchi}, \mathit{matusa}, \mathit{bunic}, \mathit{femeie}, \mathit{barbat}, \mathit{ana}, \mathit{carmen}, \mathit{elena}, \mathit{eva}, \mathit{maria}, \mathit{sara}, \mathit{valentina}, \mathit{adam}, \mathit{constantin}, \mathit{eugen}, \mathit{george}, \mathit{ion}, \mathit{iosif}, \mathit{tudor}, \mathit{victor}, 0, 1, \text{false}, \text{true})$, unde:

- $A = \{\mathit{ana}, \mathit{carmen}, \mathit{elena}, \mathit{eva}, \mathit{maria}, \mathit{sara}, \mathit{valentina}, \mathit{adam}, \mathit{constantin}, \mathit{eugen}, \mathit{george}, \mathit{ion}, \mathit{iosif}, \mathit{tudor}, \mathit{victor}\}$;
- $\mathit{stramos} \subseteq A \times A \times \mathbb{N}$ (relație ternară, predicat ternar), așadar, ca operație ternară: $\mathit{stramos} : A \times A \times \mathbb{N} \to \{\text{false}, \text{true}\}$;
- $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (operație binară);
- $\mathit{frati}, \mathit{parinte}, \mathit{tata}, \mathit{mama}, \mathit{unchi}, \mathit{matusa}, \mathit{bunic} \subseteq A \times A$ (relații binare, predicate binare), așadar, ca operații binare:
  $\mathit{frati}, \mathit{parinte}, \mathit{tata}, \mathit{mama}, \mathit{unchi}, \mathit{matusa}, \mathit{bunic} : A \times A \to \{\text{false}, \text{true}\}$;
- $\mathit{femeie}, \mathit{barbat} \subseteq A$ (relații unare, predicate unare), așadar, ca operații unare: $\mathit{femeie}, \mathit{barbat} : A \to \{\text{false}, \text{true}\}$;
- $\mathit{ana}, \mathit{carmen}, \mathit{elena}, \mathit{eva}, \mathit{maria}, \mathit{sara}, \mathit{valentina}, \mathit{adam}, \mathit{constantin}, \mathit{eugen}, \mathit{george}, \mathit{ion}, \mathit{iosif}, \mathit{tudor}, \mathit{victor} \in A$ (constante);
- $0, 1 \in \mathbb{N}$ (constante);
- $\text{false}, \text{true} \in \{\text{false}, \text{true}\}$ (constante);

operațiile nonnulare (i.e. cu argumente) ale algebrei $\mathcal{A}$ sunt definite astfel:

operația binară $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ este predefinită în Prolog;

$$femeie(x) = \begin{cases} \text{true}, & x \in \{ana, carmen, elena, eva, maria, sara, valentina\}, \\ \text{false}, & x \in A \setminus \{ana, carmen, elena, eva, maria, sara, valentina\}; \end{cases}$$

$$barbat(x) = \begin{cases} \text{true}, & x \in \{adam, constantin, eugen, george, ion, iosif, tudor, victor\}, \\ \text{false}, & x \in A \setminus \{adam, constantin, eugen, george, ion, iosif, tudor, victor\}; \end{cases}$$

pentru orice $u \in A \times A$, $parinte(u) =$

$$\begin{cases} \text{true}, & u \in \{(sara, adam), (sara, eva), (constantin, iosif), (constantin, elena), \\ & (valentina, ion), (valentina, sara), (victor, constantin), (victor, maria), \\ & (ana, victor), (ana, valentina), (carmen, victor), (carmen, valentina), \\ & (george, victor), (george, valentina), (eugen, tudor), (eugen, ana)\}, \\ \text{false}, & \text{altfel}; \end{cases}$$

pentru orice $(x, y) \in A^2$, (eliminând, conform convenției folosite uzual, o pereche de paranteze din scrierea $frati((x, y))$, și la fel mai jos)

$$frati(x, y) = \begin{cases} \text{true}, & x \neq y \text{ și } (\exists\, p \in A)\,(parinte(x, p) = parinte(y, p) = \text{true}), \\ \text{false}, & \text{altfel}; \end{cases}$$

altfel scris, cu notațiile obișnuite $\wedge$ pentru conjuncția logică și $\neg$ pentru negația logică, $frati(x, y) = \neg(x = y) \wedge \bigwedge_{p \in A} (parinte(x, p) \wedge parinte(y, p)$;

în același mod pot fi scrise definițiile următoarelor predicate: pentru orice

$(x, y) \in A^2$, $tata(x, y) = parinte(x, y) \wedge barbat(y)$,
$mama(x, y) = parinte(x, y) \wedge femeie(y)$,
$unchi(x, y) = \bigwedge_{p \in A} (parinte(x, p) \wedge frati(p, y) \wedge barbat(y))$,
$matusa(x, y) = \bigwedge_{p \in A} (parinte(x, p) \wedge frati(p, y) \wedge femeie(y))$ și
$bunic(x, y) = \bigwedge_{p \in A} (parinte(x, p) \wedge parinte(p, y))$;
iar, dacă analizăm definiția predicatului *stramos*, observăm că poate fi scrisă în modul următor: pentru orice $x, y \in A$, $stramos(x, y, 0) = \begin{cases} \text{true}, & x = y, \\ \text{false}, & x \neq y, \end{cases}$ și,
pentru orice $g \in \mathbb{N}^*$, $stramos(x, y, g) = \bigwedge (parinte(x, p) \wedge stramos(p, y, g - 1))$.

Dacă adoptăm prima soluție, a operațiilor definite parțial, atunci structura algebrică definită de programul de mai sus devine o algebră monosortată, i.e. cu o unică mulțime suport (nu familie de mulțimi ca suport):

(algebră cu unica mulțime de elemente $A \cup \mathbb{N} \cup \{\text{false}, \text{true}\}$)

$\mathcal{A}' = (A' = A \cup \mathbb{N} \cup \{\text{false}, \text{true}\})$; $stramos, +, frati, parinte, tata, mama, unchi, matusa, bunic, femeie, barbat, ana, carmen, elena, eva, maria, sara, valentina, adam, constantin, eugen, george, ion, iosif, tudor, victor, 0, 1, \text{false}, \text{true})$, unde:

- $A = \{ana, carmen, elena, eva, maria, sara, valentina, adam, constantin, eugen, george, ion, iosif, tudor, victor\}$, așadar mulțimea elementelor lui $\mathcal{A}'$ este $A' = \{ana, carmen, elena, eva, maria, sara, valentina, adam, constantin, eugen, george, ion, iosif, tudor, victor, \text{false}, \text{true}\} \cup \mathbb{N}$;
- $stramos \subseteq (A')^3 = A' \times A' \times A'$ (relație ternară, predicat ternar), așadar, ca operație ternară parțială: $stramos : (A')^3 \multimap A'$;
- $+ : A' \times A' \multimap A'$ (operație binară parțială);
- $frati, parinte, tata, mama, unchi, matusa, bunic \subseteq A' \times A'$ (relații binare, predicate binare), așadar, ca operații binare parțiale: $frati, parinte, tata, mama, unchi, matusa, bunic : A' \times A' \multimap A'$;
- $femeie, barbat \subseteq A'$ (relații unare, predicate unare), așadar, ca operații unare parțiale: $femeie, barbat : A' \multimap A'$;
- $ana, carmen, elena, eva, maria, sara, valentina, adam, constantin, eugen, george, ion, iosif, tudor, victor, 0, 1, \text{false}, \text{true} \in A'$ (constante);

operațiile nonnulare (i.e. cu argumente, de aritate nonzero) ale algebrei $\mathcal{A}'$ sunt definite ca în cazul algebrei trisortate $\mathcal{A}$, și rămân nedefinite în celelalte elemente ale domeniului lor ($(A')^2$, $A' \times A'$ sau A', după cum sunt ternare, binare, respectiv unare).

Cum răspunde Prologul la următoarea interogare?

?- $stramos(carmen, S, 3)$, $barbat(S)$.

Prologul trebuie să găsească valorile variabilei S pentru care $stramos(carmen, S, 3) \wedge barbat(S) = \text{true}$, unde $\wedge$ este operația booleană de conjuncție pe mulțimea $\{\text{false}, \text{true}\}$ a valorilor booleene predefinite în Prolog. Respectând notația pentru variabile din Prolog și fără a intra în detalii (pentru acestea a se vedea al doilea seminar), Prologul execută următorii pași:
$stramos(carmen, S, 3)$ nu unifică, cu $femeie(X)$, $barbat(X)$, $frati(X, Y)$, $parinte(X, Y)$, $tata(X, Y)$, $mama(X, Y)$, $unchi(X, Y)$, $matusa(X, Y)$ sau $bunic(X, Y)$, pentru că predicatul *stramos* nu coincide cu niciunul dintre predicatele *femeie*, *barbat*, *frati*, *parinte*, *tata*, *mama*, *unchi*, *matusa*, *bunic*;
$stramos(carmen, S, 3)$ nu unifică, cu $stramos(X, X, 0)$, pentru că, constantele 0 și 3 nu coincid;
$stramos(carmen, S, 3)$ unifică, cu $stramos(X, S, G)$ pentru $X = carmen$, așadar următorul scop compus de satisfăcut este:

?- $parinte(carmen, P)$, $stramos(P, S, H)$.

$parinte(carmen, P)$ unifică numai cu $parinte(carmen, victor)$ și $parinte(carmen, valentina)$, pentru $P = victor$, respectiv $P = valentina$.
Aceste două clauze apar ca fapte în programul Prolog de mai sus, așadar nu se intră în alte recursii după aceste unificări, ci se continuă cu satisfacerea scopurilor

$stramos(victor, S, H)$, respectiv $stramos(valentina, S, H)$ (desigur, pe rând). Acestea unifică, cu $stramos(victor, victor, 0)$, respectiv $stramos(valentina, valentina, 0)$, dar:

- $barbat(valentina)$ nu unifică, cu niciunul dintre faptele care definesc predicatul *barbat* și, desigur, cu niciun alt fapt sau membru stâng de regulă,
- în timp ce $barbat(victor)$ e satisfăcut, pentru că este unul dintre faptele din acest program, însă $0 + 1 = 1 \neq 3$, așadar $stramos(carmen, S, 3)$ nu unifică, cu $stramos(carmen, victor, 1)$.

Se continuă cu satisfacerea scopurilor $stramos(victor, S, H)$ și $stramos(valentina, S, H)$. Acestea au fost deja unificate cu clauza din faptul $stramos(X, X, 0)$; acum sunt unificate cu membrul stâng al regulii din definiția predicatului *stramos*.

Și recursia continuă în acest mod. Sunt satisfăcute $parinte(victor, constantin)$, $parinte(victor, maria)$, $parinte(valentina, ion)$ și $parinte(valentina, sara)$, precum și $stramos(constantin, constantin, 0)$, $stramos(maria, maria, 0)$, $stramos(ion, ion, 0)$ și $stramos(sara, sara, 0)$, dar nu și $barbat(maria)$ sau $barbat(sara)$, iar $0 + 1 = 1$ și $1 + 1 = 2 \neq 3$.

$parinte(maria, P)$ și $parinte(ion, P)$ nu sunt satisfăcute, pentru că acestea nu unifică, cu niciun fapt sau membru stâng al unei reguli.

Dar sunt satisfăcute $parinte(constantin, iosif)$, $parinte(constantin, elena)$, $parinte(sara, adam)$ și $parinte(sara, eva)$, precum și $stramos(iosif, iosif, 0)$, $stramos(elena, elena, 0)$, $stramos(adam, adam, 0)$ și $stramos(eva, eva, 0)$.

$barbat(elena)$ și $barbat(eva)$ nu sunt satisfăcute, dar $barbat(iosif)$ și $barbat(adam)$ sunt satisfăcute, iar $0 + 1 = 1$, $1 + 1 = 2$ și $2 + 1 = 3$.
Așadar răspunsurile (dacă le cerem pe amândouă, cu ";"/"Next") Prologului la interogarea:

?- $stramos(carmen, S, 3)$, $barbat(S)$.

vor fi:

$S = iosif$ și $S = adam$.

Cum sunt efectuate unificările de mai sus?
Prologul aplică **algoritmul de unificare**.
Amintesc că și **constantele** sunt **operații**, anume *operații fără argumente*, *operații cu 0 argumente*, *operații de aritate 0*, numite și *operații zeroare* sau *nulare*, așadar nu trebuie tratate ca un caz separat în cele ce urmează.
Și, referitor la terminologia din secțiunea recapitulativă care deschide acest curs, pentru că **predicatele**, i.e. **relațiile**, sunt considerate OPERAŢII AVÂND REZULTATUL BOOLEAN, **formulele atomice** vor fi considerate tot **termeni**, având OPERAŢIA DOMINANTĂ DE REZULTAT BOOLEAN.
În esență, pentru orice $n, k \in \mathbb{N}$, orice operații (incluzând aici predicatele, relațiile) f și g de arități n, respectiv k și orice termeni (care pot avea operația dominantă de rezultat boolean, i.e. predicat, relație) $t_1, \ldots, t_n, u_1, \ldots, u_k$:

(în esență, algoritmul de unificare funcționează astfel)

termenii $f(t_1, \ldots, t_n)$ și $g(u_1, \ldots, u_k)$ unifică ddacă: $\begin{cases} n = k, \\ t_1 \text{ și } u_1 \text{ unifică}, \\ \vdots \\ t_n \text{ și } u_n \text{ unifică}, \end{cases}$

și această recursie continuă până la unificări de termeni cu termeni care sunt:

- fie variabile, iar acestea unifică, cu ele însele și cu orice termen care nu le conține,
- fie constante, iar acestea sunt operații fără argumente, așadar, conform regulii de mai sus, nu unifică decât cu ele însele și cu orice variabilă.

Exemplu

$stramos(carmen, S, 3)$:

- nu unifică, cu $bunic(X, Y)$, pentru că $stramos \neq bunic$;
- nu unifică, cu $stramos(X, X, 0)$, pentru că $3 \neq 0$;
- unifică, cu $stramos(X, X, Y)$ pentru $carmen = X = S$ și $Y = 3$;
- nu unifică, cu $stramos(X, X, X)$ pentru că, dacă ar avea loc această unificare, atunci am avea unificările $carmen = X$, $S = X$ și $3 = X$, deci $X = carmen = S = 3$, așadar constantele $carmen$ și 3 ar trebui să unifice, dar

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Să adaptăm cadrul de lucru din prima secțiune a acestui curs la cadrul de lucru pentru programarea în Prolog: vom considera o **structură de ordinul I** $\mathcal{A}$, fie multisortată, fie cu operațiile parțial definite, și în care relațiile (predicatele) sunt considerate operații având rezultatul **true** sau **false**, iar constantele sunt operațiile zeroare, astfel că $\mathcal{A}$ este înzestrată doar cu o familie $(f_i)_{i \in I}$ de operații.

Notație

Pentru fiecare $i \in I$, să notăm cu $dom(f_i)$ domeniul funcției f_i, adică:

1. dacă, pentru o mulțime nevidă S, $\mathcal{A}$ este o algebră S–sortată cu mulțimea suport $A = (A_s)_{s \in S}$ (mulțime S–sortată, adică familie de mulțimi indexată de S), atunci, pentru fiecare $i \in I$, există $n_i \in \mathbb{N}$ și $s_{i,1}, \ldots, s_{i,n_i}, s_i \in S$ astfel încat $f_i : A_{s_{i,1}} \times \ldots \times A_{s_{i,n_i}} \to A_{s_i}$; atunci $\underline{\underline{dom(f_i) = A_{s_{i,1}} \times \ldots \times A_{s_{i,n_i}}}}$;
2. dacă $\mathcal{A}$ este o algebră monosortată cu mulțimea suport A (monosortată, i.e. o unică mulțime) și cu operațiile parțial definite, atunci, pentru fiecare $i \in I$, există $n_i \in \mathbb{N}$ astfel încat $f_i : A^{n_i} \multimap A$; atunci $\underline{\underline{dom(f_i)}}$ este mulțimea elementelor produsului cartezian A^{n_i} în care funcția parțială f_i este definită.

Notăm cu $dom(\mathcal{A})$ mulțimea elementelor lui $\mathcal{A}$, indiferent de sort/tip de date, i.e.:

1. în cazul S–sortat de mai sus, $dom(\mathcal{A}) = \bigcup_{s \in S} A_s$;
2. în cazul monosortat de mai sus, $dom(\mathcal{A}) = A$.

Observație (putem pune tipurile de date laolaltă, la fel ca în Prolog, unde toate sunt înglobate în tipul TERMEN)

Cazul multisortat poate fi înglobat în cazul monosortat cu operații parțiale, astfel: pentru orice $i \in I$, dacă f_i are aritatea $n_i \in \mathbb{N}^*$, atunci considerăm

$$f_i : dom(\mathcal{A})^{n_i} \multimap dom(\mathcal{A}),$$

f_i definită numai pe elementele lui $dom(f_i)$.

În cele ce urmează, vom lucra cu algebre monosortate înzestrate cu operații parțiale.

Definiție

Tipul sau *signatura* structurii algebrice $\mathcal{A}$ este $\tau = (n_i)_{i \in I} \subseteq \mathbb{N}$, unde, pentru fiecare $i \in I$, n_i este aritatea lui f_i.

Pentru fiecare $i \in I$, vom considera un **simbol de operație** n_i–ară φ_i. Elementele mulțimii $\{\varphi_i \mid i \in I\}$ le considerăm **două câte două distincte**.

Definiție

O *structură algebrică de signatură* τ este o algebră $\mathcal{M} = (M; (\varphi_i^{\mathcal{M}})_{i \in I})$, unde, pentru fiecare $i \in I$, $\varphi_i^{\mathcal{M}} : M^{n_i} \multimap M$.

Exemplu

Astfel, algebra $\mathcal{A}$ de mai sus este algebra de signatură τ $(dom(\mathcal{A}); (\varphi_i^{\mathcal{A}})_{i \in I})$, unde, pentru fiecare $i \in I$, $\varphi_i^{\mathcal{A}} = f_i$.

Să considerăm și o mulțime infinită *Var* de **variabile**, ale cărei elemente le considerăm **două câte două distincte**.
De asemenea, considerăm mulțimea de variabile *disjunctă* de cea a simbolurilor de operații: $Var \cap \{\varphi_i \mid i \in I\} = \emptyset$.
Deocamdată, vom considera doar **alfabetul** format din **variabile**, **simboluri de operații**, virgula și parantezele rotunde: $Var \cup \{\varphi_i \mid i \in I\} \cup \{,\} \cup \{(,)\}$.

Definiție și notație (termenii peste alfabetul de mai sus)

Considerăm următoarea mulțime de cuvinte finite și nevide peste alfabetul de mai sus: $Term \subseteq (Var \cup \{\varphi_i \mid i \in I\} \cup \{,\} \cup \{(,)\})^+ = \{a_1 \ldots a_n \mid n \in \mathbb{N}^*, a_1, \ldots, a_n \in Var \cup \{\varphi_i \mid i \in I\} \cup \{,\} \cup \{(,)\}\}$, ale cărei elemente le vom numi *termeni*, definită recursiv în modul următor:

1. $Var \subseteq Term$ (variabilele sunt termeni de lungime 1, i.e. constând dintr–o singură literă: variabila respectivă);
2. pentru orice $i \in I$, dacă $n_i = 0$ (adică φ_i este un *simbol de operație zeroară*, un *simbol de constantă*, atunci $\varphi_i \in Term$ (simbolurile de constante sunt termeni, tot de lungime 1, constând dintr–o singură literă, anume acel simbol de constantă);
3. pentru orice $i \in I$, dacă $n_i \in \mathbb{N}^*$ (adică φ_i este un simbol de operație cu $n_i \neq 0$ argumente), atunci, pentru orice $t_1, \ldots, t_{n_i} \in Term$, rezultă că $\varphi_i(t_1, \ldots, t_{n_i}) \in Term$ (orice simbol de operație cu argumente aplicată unor termeni are drept rezultat un termen); termenii formați în acest al treilea mod se numesc *termeni compuși*.

Observație

Cum variabilele și simbolurile de operații sunt două câte două distincte, rezultă că orice termen are o unică scriere ca:

- variabilă,
- simbol de constantă sau
- termen compus, de forma $\varphi(t_1, \ldots, t_n)$, cu simbolul de operație φ, $n \in \mathbb{N}^*$ și termenii $t_1, \ldots, t_n$ unic determinați.

Definiție și notație

Pentru orice termen $t \in Term$, notăm cu $V(t)$ *mulțimea variabilelor care apar în* t, definită, recursiv, astfel:

- oricare ar fi $v \in Var$, $V(v) = \{v\}$;
- pentru orice $i \in I$ cu $n_i = 0$, $V(\varphi_i) = \emptyset$ (i.e. simbolurile de constante nu au variabile);
- pentru orice $i \in I$ cu $n_i \in \mathbb{N}^*$ și orice termeni $t_1, \ldots, t_{n_i} \in Term$, $V(\varphi_i(t_1, \ldots, t_{n_i})) = V(t_1) \cup \ldots \cup V(t_{n_i})$.

Definiție (substituțiile dau variabilelor valori în mulțimea termenilor)

O *substituție* este o funcție $\sigma : Var \to Term$.

Notație

Dacă $n \in \mathbb{N}^*$, $\{v_1, \ldots, v_n\} \subseteq Var$ este o mulțime finită de variabile, iar $\{t_1, \ldots, t_n\} \subseteq Term$ este o mulțime de termeni, atunci se notează cu $\{v_1/t_1, \ldots, v_n/t_n\}$ următoarea substituție:

$$\{v_1/t_1, \ldots, v_n/t_n\} : Var \to Term, \text{ pentru orice } v \in Var,$$

$$\{v_1/t_1, \ldots, v_n/t_n\}(v) = \begin{cases} t_i, & (\exists\, i \in \overline{1, n})\,(v = v_i), \\ v, & v \in Var \setminus \{v_1, \ldots, v_n\}. \end{cases}$$

Definiție și notație

Fie $\sigma : Var \to Term$ o substituție. Următoarea extindere a lui σ la întreaga mulțime a termenilor se numește tot *substituție* (și, de obicei, se notează tot cu σ): $\widetilde{\sigma} : Term \to Term$, definită, recursiv, astfel:

- pentru orice $v \in Var$, $\widetilde{\sigma}(v) = \sigma(v)$ (adică $\widetilde{\sigma}\,|_{Var} = \sigma$: pe variabile, $\widetilde{\sigma}$ coincide cu σ);
- pentru orice $i \in I$ astfel încât $n_i = 0$, $\widetilde{\sigma}(\varphi_i) = \varphi_i$ (adică, pe simbolurile de constante, $\widetilde{\sigma}$ este funcția identică);
- pentru orice $i \in I$ astfel încât $n_i \in \mathbb{N}^*$ și orice termeni $t_1, \ldots, t_{n_i} \in Term$ (astfel încât $\widetilde{\sigma}$ a fost definită în $t_1, \ldots, t_{n_i}$), $\widetilde{\sigma}(\varphi_i(t_1, \ldots, t_{n_i})) = \varphi_i(\widetilde{\sigma}(t_1), \ldots, \widetilde{\sigma}(t_n))$.

Observație

Extinderea $\widetilde{\sigma}$ din definiția anterioară este:

- **complet definită**, pentru că toți termenii se scriu ca mai sus, i.e. sunt obținuți prin acea recursie;
- **corect definită**, pentru că orice termen are o unică scriere ca mai sus.

Definiție (să dăm variabilelor valori într–o algebră de signatură τ, apoi să calculăm valorile tuturor termenilor pe baza celor date variabilelor – reținem această definiție pentru mai târziu)

Pentru orice algebră $\mathcal{M} = (M; (\varphi_i^{\mathcal{M}})_{i \in I})$ de signatură τ, o funcție $s : Var \to M$ va fi numită *interpretare*.
Următoarea prelungire a lui s la mulțimea *Term* a tuturor termenilor se numește tot *interpretare* și se notează, de obicei, tot cu s: $\widetilde{s} : Term \to M$, definită, recursiv, astfel:

- pentru orice $v \in Var$, $\widetilde{s}(v) = s(v)$ (i.e. $\widetilde{s}\,|_{Var} = s$);
- pentru orice $i \in I$ cu $n_i = 0$, $\widetilde{s}(\varphi_i) = \varphi_i^{\mathcal{M}}$ (valoarea lui $\widetilde{s}$ într–un simbol de constantă este constanta din $\mathcal{M}$ corespunzătoare acelui simbol de constantă);
- pentru orice $i \in I$ cu $n_i \in \mathbb{N}^*$ și orice $t_1, \ldots, t_{n_i} \in Term$ astfel încât ($\widetilde{s}$ a fost definită în termenii $t_1, \ldots, t_{n_i}$ și) $(\widetilde{s}(t_1), \ldots, \widetilde{s}(t_{n_i})) \in dom(\varphi_i^{\mathcal{M}})$, $\widetilde{s}(\varphi_i(t_1, \ldots, t_{n_i})) = \varphi_i^{\mathcal{M}}(\widetilde{s}(t_1), \ldots, \widetilde{s}(t_{n_i}))$.

Algoritm pentru rezolvarea unei probleme de unificare

Definiție (ce înseamnă a unifica mai mulți termeni)

Fie $k \in \mathbb{N} \setminus \{0, 1\}$ și $t_1, \ldots, t_k \in \mathit{Term}$. Spunem că termenii $t_1, \ldots, t_k$ *unifică* ddacă există o substituție $\sigma : \mathit{Term} \to \mathit{Term}$, numită *unificator pentru* $t_1, \ldots, t_k$, cu proprietatea că $\sigma(t_1) = \ldots = \sigma(t_k)$.

Fixăm un număr natural $k \geq 2$ și k termeni $t_1, \ldots, t_k \in \mathit{Term}$.

Cerința de a determina dacă termenii $t_1, \ldots, t_k$ unifică și, în caz afirmativ, a determina și un unificator pentru acești termeni este o *problemă de unificare*.

Observație

În cazul în care $t_1, \ldots, t_k$ unifică, următorul algoritm obține, într–un număr finit de pași, *un cel mai general unificator pentru* $t_1, \ldots, t_k$, adică un unificator $\mu : \mathit{Term} \to \mathit{Term}$ cu proprietatea că orice unificator $\sigma : \mathit{Term} \to \mathit{Term}$ pentru $t_1, \ldots, t_k$ este de forma $\sigma = \lambda \circ \mu$ pentru o substituție $\lambda : \mathit{Term} \to \mathit{Term}$.
În cazul în care $t_1, \ldots, t_k$ nu unifică, algoritmul următor se termină într–un număr finit de pași cu EŞEC, i.e. fără a găsi un unificator.

(Algoritmul de unificare)

Vom reține două liste (mulțimi) de *ecuații* (*egalități*, *probleme de unificare între câte doi termeni*):

o *listă soluție* S și o *listă de rezolvat* R.

Inițial:

- $S = \emptyset$;
- $R = \{t_1 = t_2, t_2 = t_3, \ldots, t_{k-1} = t_k\}$.

Se aplică, pe rând, următorii pași, în orice ordine posibilă:

- SCOATERE (ELIMINARE, ŞTERGERE): orice ecuație de forma $t = t$ (i.e. între un termen și el însuși) este eliminată din lista R;
- DESCOMPUNERE: orice ecuație de forma $\varphi_i(u_1, \ldots, u_{n_i}) = \varphi_i(w_1, \ldots, w_{n_i})$, unde $i \in I$ astfel încât $n_i > 0$ și $u_1, \ldots, u_{n_i}, w_1, \ldots, w_{n_i} \in Term$ din lista R este înlocuită cu următoarele n_i ecuații: $u_1 = w_1, \ldots, u_{n_i} = w_{n_i}$;
- REZOLVARE: orice ecuație din R de forma $v = t$ sau $t = v$, unde $v \in Var$ și $t \in Term$, este scoasă din R și introdusă în lista soluție S sub forma $v = t$ (dacă și $t \in Var$, atunci nu contează cum orientăm ecuația: putem alege pe oricare dintre variabile ca membru stâng), apoi toate aparițiile lui v în termenii care apar în ecuațiile din lista de rezolvat R și din lista soluție S se înlocuiesc cu t, adică toți acești termeni u se înlocuiesc cu $\{v/t\}(u)$.

Algoritmul se ÎNCHEIE în oricare dintre cazurile următoare:

1. dacă, după execuția unui pas de SCOATERE sau REZOLVARE, obținem $R = \emptyset$, i.e. lista de rezolvat devine vidă, iar lista soluție curentă este $S = \{v_1 = u_1, \ldots, v_n = u_n\}$, atunci **un (cel mai general) unificator** pentru $t_1, \ldots, t_k$ este substituția $\{v_1/u_1, \ldots, v_n/u_n\}$: IEŞIRE CU SUCCES;
2. dacă, în cursul execuției pașilor de mai sus, apare în lista de rezolvat R:
   - fie o ecuație de forma $v = t$, cu $t \in Term$ și $v \in V(t)$ (i.e. cu variabila v apărând în t),
   - fie o ecuație de forma $\varphi_i(u_1, \ldots, u_{n_i}) = \varphi_j(w_1, \ldots, w_{n_j})$, cu $i, j \in I$, $n_i, n_j \in \mathbb{N}$ (nu neapărat nenule), $u_1, \ldots, u_{n_i}, w_1, \ldots, w_{n_j} \in Term$ și $i \neq j$, așadar cu simbolurile de operații dominante ale termenilor din cei doi membri diferite: $\varphi_i \neq \varphi_j$,

   atunci se IESE CU EŞEC: nu s–a găsit niciun unificator, așadar **termenii** $t_1, \ldots, t_k$ **nu unifică**.

Exercițiu (temă pentru seminar)

Să considerăm două simboluri de operații binare distincte f și g, unul de operație unară h, două simboluri de constante diferite a și b și patru variabile $V, X, Y, Z \in Var$, două câte două distincte.
Considerăm următorii termeni formați cu simbolurile de operații și variabilele de mai sus: $r, s, t, u, w \in Term$,

Exercițiu (continuare: termenii de unificat, dați prin expresiile lor și prin arborii asociați acestor expresii)

$r = f(h(h(a)), g(g(X, h(Y)), X))$,
$s = f(h(X), g(g(X, X), h(Y)))$,
$t = f(h(V), g(Z, Z))$,
$u = f(h(h(Z)), g(g(X, h(b)), h(Z)))$,
$w = f(h(b), g(g(V, Z), f(V, Z)))$.

Să se unifice acești termeni doi câte doi, adică să se rezolve, pe rând, problemele de unificare:

$r = s, r = t,$
$r = u, r = w,$
$s = t, s = u,$
$s = w, t = u,$
$t = w, u = w.$

Pentru perechile $\{p, q\}$ de termeni $p, q \in \{r, s, t, u, w\}$ care unifică, să se unifice și reuniunea tuturor acestor perechi.

Rezolvare: Aplicăm ALGORITMUL DE UNIFICARE.

În continuare, simbolurile de operații vor fi numite, simplu, *operații*, în particular simbolurile de constante vor fi numite, simplu, *constante*.

① Rezolvăm **problema de unificare** $r = s$, adică
$f(h(h(a)), g(g(X, h(Y)), X)) = f(h(X), g(g(X, X), h(Y)))$.
INIŢIALIZARE: lista soluție $S = \emptyset$, lista de rezolvat
$R = \{f(h(h(a)), g(g(X, h(Y)), X)) = f(h(X), g(g(X, X), h(Y)))\}$.
DESCOMPUNERE (în operanzii lui f: operator binar): $S = \emptyset$,
$R = \{h(h(a)) = h(X), g(g(X, h(Y)), X) =$
$g(g(X, X), h(Y)))\}$.
DESCOMPUNERE (în operanzii lui h: operator unar): $S = \emptyset$,
$R = \{h(a) = X, g(g(X, h(Y)), X) = g(g(X, X), h(Y)))\}$.
REZOLVARE: $S = \{X = h(a)\}$,
$R = \{g(g(h(a), h(Y)), h(a)) = g(g(h(a), h(a)), h(Y)))\}$.
DESCOMPUNERE (în operanzii lui g: operator binar): $S = \{X = h(a)\}$,
$R = \{g(h(a), h(Y)) = g(h(a), h(a)), h(a) = h(Y)\}$.
DESCOMPUNERE (în operanzii lui h: operator unar): $S = \{X = h(a)\}$,
$R = \{g(h(a), h(Y)) = g(h(a), h(a)), a = Y\}$.
REZOLVARE: $S = \{X = h(a), Y = a\}$, $R = \{g(h(a), h(a)) = g(h(a), h(a))\}$.
SCOATERE: $S = \{X = h(a), Y = a\}$, $R = \emptyset$.
Cum $R = \emptyset$, se IESE CU SUCCES, cu **unificatorul** pentru termenii r și s dat de lista S, anume substituția $\{X/h(a), Y/a\}$.

② Rezolvăm **problema de unificare** $r = t$, adică
$f(h(h(a)), g(g(X, h(Y)), X)) = f(h(V), g(Z, Z))$.
INIŢIALIZARE: $S = \emptyset$, $R = \{f(h(h(a)), g(g(X, h(Y)), X)) = f(h(V), g(Z, Z))\}$.
DESCOMPUNERE (în operanzii lui f): $S = \emptyset$,
$R = \{h(h(a)) = h(V), g(g(X, h(Y)), X) = g(Z, Z)\}$.
DESCOMPUNERE (în operanzii lui g: tot operator binar): $S = \emptyset$,
$R = \{h(h(a)) = h(V), g(X, h(Y)) = Z, X = Z\}$.
REZOLVARE (ambii membri ai ecuației introduse în S sunt variabile, așa că o putem alege pe oricare ca membru stang): $S = \{X = Z\}$,
$R = \{h(h(a)) = h(V), g(Z, h(Y)) = Z\}$.
Cum $Z \in V(g(Z, h(Y)))$ (variabila Z apare în termenul $g(Z, h(Y))$), se IESE CU EŞEC: termenii r și t **nu unifică**, **nu au unificator**.

③ Rezolvăm **problema de unificare** $r = u$, adică
$f(h(h(a)), g(g(X, h(Y)), X)) = f(h(h(Z)), g(g(X, h(b)), h(Z)))$.
INIŢIALIZARE: $S = \emptyset$,
$R = \{f(h(h(a)), g(g(X, h(Y)), X)) = f(h(h(Z)), g(g(X, h(b)), h(Z)))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(h(a)) = h(h(Z)), g(g(X, h(Y)), X) = g(g(X, h(b)), h(Z))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(a) = h(Z), g(g(X, h(Y)), X) = g(g(X, h(b)), h(Z))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{a = Z, g(g(X, h(Y)), X) = g(g(X, h(b)), h(Z))\}$.
REZOLVARE: $S = \{Z = a\}$, $R = \{g(g(X, h(Y)), X) = g(g(X, h(b)), h(a))\}$.
DESCOMPUNERE: $S = \{Z = a\}$, $R = \{g(X, h(Y)) = g(X, h(b)), X = h(a)\}$.
REZOLVARE: $S = \{X = h(a), Z = a\}$, $R = \{g(h(a), h(Y)) = g(h(a), h(b))\}$.
DESCOMPUNERE: $S = \{X = h(a), Z = a\}$, $R = \{h(a) = h(a), h(Y) = h(b)\}$.
SCOATERE: $S = \{X = h(a), Z = a\}$, $R = \{h(Y) = h(b)\}$.
DESCOMPUNERE: $S = \{X = h(a), Z = a\}$, $R = \{Y = b\}$.
REZOLVARE: $S = \{X = h(a), Y = b, Z = a\}$, $R = \emptyset$.
Cum $R = \emptyset$, se IESE CU SUCCES: r și u **au unificatorul** $\{X/h(a), Y/b, Z/a\}$.

④ Rezolvăm **problema de unificare** $r = w$, adică
$f(h(h(a)), g(g(X, h(Y)), X)) = f(h(h(Z)), g(g(Y, X), f(Y, X)))$.
INIŢIALIZARE: $S = \emptyset$,
$R = \{f(h(h(a)), g(g(X, h(Y)), X)) = f(h(b), g(g(Y, X), f(Y, X)))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(h(a)) = h(b), g(g(X, h(Y)), X) = g(g(Y, X), f(Y, X))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(a) = b, g(g(X, h(Y)), X) = g(g(Y, X), f(Y, X))\}$.
Cum $h \neq b$ (operația unară h nu coincide cu operația zeroară, i.e. constanta b), se IESE CU EŞEC: r și w **nu au unificator**.

⑤ Rezolvăm **problema de unificare** $s = t$, adică
$f(h(X), g(g(X, X), h(Y))) = f(h(V), g(Z, Z))$.
INIŢIALIZARE: $S = \emptyset$, $R = \{f(h(X), g(g(X, X), h(Y))) = f(h(V), g(Z, Z))\}$.
DESCOMPUNERE: $S = \emptyset$, $R = \{h(X) = h(V), g(g(X, X), h(Y)) = g(Z, Z)\}$.
DESCOMPUNERE: $S = \emptyset$, $R = \{h(X) = h(V), g(X, X) = Z, h(Y) = Z\}$.
REZOLVARE: $S = \{Z = h(Y)\}$, $R = \{h(X) = h(V), g(X, X) = h(Y)\}$.
Cum $g \neq h$ (operațiile g și h nu coincid), se IESE CU EŞEC: s și t **nu unifică**.

⑥ Rezolvăm **problema de unificare** $s = u$, adică
$f(h(X), g(g(X, X), h(Y))) = f(h(h(Z)), g(g(X, h(b)), h(Z)))$.
INIŢIALIZARE: $S = \emptyset$,
$R = \{f(h(X), g(g(X, X), h(Y))) = f(h(h(Z)), g(g(X, h(b)), h(Z)))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(X) = h(h(Z)), g(g(X, X), h(Y)) = g(g(X, h(b)), h(Z))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{X = h(Z), g(g(X, X), h(Y)) = g(g(X, h(b)), h(Z))\}$.
REZOLVARE: $S = \{X = h(Z)\}$,
$R = \{g(g(h(Z), h(Z)), h(Y)) = g(g(h(Z), h(b)), h(Z))\}$.
DESCOMPUNERE: $S = \{X = h(Z)\}$,
$R = \{g(h(Z), h(Z)) = g(h(Z), h(b)), h(Y) = h(Z)\}$.
DESCOMPUNERE: $S = \{X = h(Z)\}$,
$R = \{g(h(Z), h(Z)) = g(h(Z), h(b)), Y = Z\}$.
REZOLVARE: $S = \{X = h(Z), Y = Z\}$, $R = \{g(h(Z), h(Z)) = g(h(Z), h(b))\}$.
DESCOMPUNERE: $S = \{X = h(Z), Y = Z\}$, $R = \{h(Z) = h(Z), h(Z) = h(b)\}$.
SCOATERE: $S = \{X = h(Z), Y = Z\}$, $R = \{h(Z) = h(b)\}$.
DESCOMPUNERE: $S = \{X = h(Z), Y = Z\}$, $R = \{Z = b\}$.
REZOLVARE: $S = \{X = h(b), Y = b, Z = b\}$, $R = \emptyset$.
Cum $R = \emptyset$, se IESE CU SUCCES: s și u **au unificatorul** $\{X/h(b), Y/b, Z/b\}$.

⑦ Rezolvăm **problema de unificare** $s = w$, adică
$f(h(X), g(g(X, X), h(Y))) = f(h(b), g(g(Y, X), f(Y, X)))$.
INIŢIALIZARE: $S = \emptyset$,
$R = \{f(h(X), g(g(X, X), h(Y))) = f(h(b), g(g(Y, X), f(Y, X)))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(X) = h(b), g(g(X, X), h(Y)) = g(g(Y, X), f(Y, X))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(X) = h(b), g(X, X) = g(Y, X), h(Y) = f(Y, X)\}$.
Cum $h \neq f$ (operațiile h și f nu coincid), se IESE CU EŞEC: s și w **nu unifică**.

⑧ Rezolvăm **problema de unificare** $t = u$, adică
$f(h(V), g(Z, Z)) = f(h(h(Z)), g(g(X, h(b)), h(Z)))$.
INIŢIALIZARE: $S = \emptyset$,
$R = \{f(h(V), g(Z, Z)) = f(h(h(Z)), g(g(X, h(b)), h(Z)))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(V) = h(h(Z)), g(Z, Z) = g(g(X, h(b)), h(Z))\}$.
DESCOMPUNERE: $S = \emptyset$, $R = \{h(V) = h(h(Z)), Z = g(X, h(b)), Z = h(Z)\}$.
Cum $Z \in V(h(Z))$ (variabila Z apare în termenul $h(Z)$), se IESE CU EŞEC: t și u **nu unifică**.

⑨ Rezolvăm **problema de unificare** $t = w$, adică
$f(h(V), g(Z, Z)) = f(h(b), g(g(Y, X), f(Y, X)))$.
INIŢIALIZARE: $S = \emptyset$, $R = \{f(h(V), g(Z, Z)) = f(h(b), g(g(Y, X), f(Y, X)))\}$.
DESCOMPUNERE: $S = \emptyset$, $R = \{h(V) = h(b), g(Z, Z) = g(g(Y, X), f(Y, X))\}$.
DESCOMPUNERE: $S = \emptyset$, $R = \{h(V) = h(b), Z = g(Y, X), Z = f(Y, X)\}$.
REZOLVARE: $S = \{Z = f(Y, X)\}$, $R = \{h(V) = h(b), f(Y, X) = g(Y, X)\}$.
Cum $f \neq g$ (operațiile binare f și g nu coincid), se IESE CU EŞEC: t și w **nu unifică**.

⑩ Rezolvăm **problema de unificare** $u = w$, adică
$f(h(h(Z)), g(g(X, h(b)), h(Z))) = f(h(b), g(g(Y, X), f(Y, X)))$.
INIŢIALIZARE: $S = \emptyset$,
$R = \{f(h(h(Z)), g(g(X, h(b)), h(Z))) = f(h(b), g(g(Y, X), f(Y, X)))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(h(Z)) = h(b), g(g(X, h(b)), h(Z)) = g(g(Y, X), f(Y, X))\}$.
DESCOMPUNERE: $S = \emptyset$,
$R = \{h(h(Z)) = h(b), g(X, h(b)) = g(Y, X), h(Z) = f(Y, X)\}$.
Cum $h \neq f$ (operațiile h și f nu coincid), se IESE CU EŞEC: u și w **nu unifică**.

⑪ Conform celor de mai sus, perechea r și s, perechea r și u și perechea s și u unifică, iar celelalte perechi de termeni nu unifică. Așadar, pentru ultima cerință a exercițiului, avem de unificat termenii r, s și u.

• *Prima metodă:* Pentru a unifica termenii r, s și u, putem aplica încă o dată ALGORITMUL DE UNIFICARE, pentru a rezolva **problema de unificare** $\{r = s, s = u\}$, i.e. $\{f(h(h(a)), g(g(X, h(Y)), X)) = f(h(X), g(g(X, X), h(Y))),$ $f(h(X), g(g(X, X), h(Y))) = f(h(h(Z)), g(g(X, h(b)), h(Z)))\}$. Putem "refolosi" pași celor două aplicări ale algoritmului de unificare pentru unificările $r = s$ și $s = u$; putem aplica mai mulți pași de DESCOMPUNERE și SCOATERE simultan, dar nu mai mulți pași de REZOLVARE simultan.
INIȚIALIZARE: $S = \emptyset$, $R = \{f(h(h(a)), g(g(X, h(Y)), X)) =$ $f(h(X), g(g(X, X), h(Y))), f(h(X), g(g(X, X), h(Y)))$ $= f(h(h(Z)), g(g(X, h(b)), h(Z)))\}$.
Conform primilor doi pași de DESCOMPUNERE din fiecare dintre unificările $r = s$ și $s = u$, obținem: $S = \emptyset$, $R = \{h(a) = X, g(g(X, h(Y)), X) =$ $g(g(X, X), h(Y))), X = h(Z), g(g(X, X), h(Y)) = g(g(X, h(b)), h(Z))\}$.
REZOLVARE: $S = \{X = h(a)\}$,
$R = \{g(g(h(a), h(Y)), h(a)) = g(g(h(a), h(a)), h(Y))), h(a) = h(Z),$ $g(g(h(a), h(a)), h(Y)) = g(g(h(a), h(b)), h(Z))\}$.

DESCOMPUNERE: $S = \{X = h(a)\}$,
$R = \{g(g(h(a), h(Y)), h(a)) = g(g(h(a), h(a)), h(Y))), a = Z,$
$g(g(h(a), h(a)), h(Y)) = g(g(h(a), h(b)), h(Z))\}$.
REZOLVARE: $S = \{X = h(a), Z = a\}$,
$R = \{g(g(h(a), h(Y)), h(a)) = g(g(h(a), h(a)), h(Y))), g(g(h(a), h(a)), h(Y))$
$= g(g(h(a), h(b)), h(a))\}$.
Doi pași de DESCOMPUNERE: $S = \{X = h(a), Z = a\}$,
$R = \{g(h(a), h(Y)) = g(h(a), h(a)), h(a) = h(Y),$
$g(h(a), h(a)) = g(h(a), h(b)), h(Y) = h(a)\}$.
DESCOMPUNERE: $S = \{X = h(a), Z = a\}$, $R = \{g(h(a), h(Y)) =$
$g(h(a), h(a)), h(a) = h(Y), h(a) = h(a), h(a) = h(b), h(Y) = h(a)\}$.
DESCOMPUNERE: $S = \{X = h(a), Z = a\}$, $R = \{g(h(a), h(Y)) =$
$g(h(a), h(a)), h(a) = h(Y), h(a) = h(a), a = b, h(Y) = h(a)\}$.
Cum $a \neq b$ (simbolurile de constante a și b nu coincid), se IESE CU EŞEC: termenii r, s și u **nu unifică**.

• *A doua metodă:* Conform celor de mai sus și proprietăților ALGORITMULUI DE UNIFICARE, avem următoarele:
un cel mai general unificator pentru r și s este substituția $\{X/h(a), Y/a\}$, pe care o notăm $\rho : Term \to Term$,
un cel mai general unificator pentru r și u este $\{X/h(a), Y/b, Z/a\}$,

iar **un cel mai general unificator** pentru s și u este $\{X/h(b), Y/b, Z/b\}$, pe care îl notăm $\sigma : Term \to Term$.
Substituțiile de mai sus nu sunt **nici singurii unificatori**, **nici singurii cei mai generali unificatori** ai acestor perechi de termeni. Însă, conform definiției unui *cel mai general unificator*, avem că:
orice unificator pentru r și s este o compunere a unei alte substituții cu unificatorul ρ,
și **orice unificator** pentru s și u este o compunere a unei alte substituții cu σ.
Desigur, la fel pentru r și u și unificatorul de mai sus.
Așadar, dacă o substituție $\mu : Term \to Term$ este un unificator pentru r, s și u, atunci există substituții $\kappa : Term \to Term$ și $\lambda : Term \to Term$ astfel încât $\mu = \kappa \circ \rho = \lambda \circ \sigma$.
Rezultă că:
$\mu(X) = \kappa(\rho(X)) = \kappa(h(a)) = h(\kappa(a)) = h(a)$ și
$\mu(X) = \lambda(\sigma(X)) = \lambda(h(b)) = h(\lambda(b)) = h(b)$,
de unde rezultă că $h(a) = h(b)$; avem o contradicție, pentru că $a \neq b$ (constantele a și b diferă), așadar $h(a) \neq h(b)$: termenii $h(a)$ și $h(b)$ sunt diferiți; amintesc că termenii sunt cuvinte peste alfabetul acestui limbaj de ordinul I, așadar doi termeni coincid ddacă sunt LITERAL IDENTICI, i.e. de aceeași lungime și formați din aceleași litere.
Rezultă că nu există o substituție μ care să unifice termenii r, s și u, adică **nu există unificator** pentru r, s și u, i.e. termenii r, s și u **nu unifică**.

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

În primul rând, ce tip de signatură avem în Prolog și cu ce fel de structuri algebrice de acea signatură lucrăm?
Am observat că avem nevoie de **operații parțiale**, fiecare având în *domeniul de definiție* elemente de un anumit **tip de date** (sau de mai multe tipuri de date).
Toate tipurile de date sunt înglobate în **tipul termen**, adică sunt **subtipuri** ale tipului termen.
Să considerăm o algebră $\mathcal{A} = (A; (f_i)_{i \in I}; (R_j)_{j \in J}; (c_k)_{k \in K})$ de o signatură $\tau = ((n_i)_{i \in I}; (m_j)_{j \in J}; (0)_{k \in K})$.
Cum trebuie să arate algebra $\mathcal{A}$ ca să corespundă unui program în Prolog?
Predicatele vor fi reprezentate prin **relațiile** din familia de relații $(R_j)_{j \in J}$. Știm că, în Prolog, putem imbrica predicatele, așadar avem nevoie de următoarele:

- mulțimea A trebuie să conțină constantele booleene **true** și **false**;
- o parte dintre operațiile parțiale din familia $(f_i)_{i \in I}$ sunt asociate relațiilor din familia $(R_j)_{j \in J}$, astfel:
  considerăm $J \subseteq I$;
  pentru fiecare $j \in J$, avem: $n_j = m_j$ și $f_j : A^{n_j} \multimap A$ (cu $Im(f_j) \subseteq \{\text{true}, \text{false}\}$, unde, prin **definiție**, imaginea funcției parțiale f_j coincide cu imaginea funcției $f_j \mid_{dom(f_j)}: dom(f_j) \to A$), definită prin: oricare ar fi $(a_1, \ldots, a_{n_j}) \in dom(f_j)$,
  $$f(a_1, \ldots, a_{n_j}) = \begin{cases} \text{true}, & \text{dacă } (a_1, \ldots, a_{n_j}) \in R_j; \\ \text{false}, & \text{altfel}; \end{cases}$$

în practică, vom nota funcțiile f_j la fel ca pe relațiile (predicatele) R_j;

- iar **conectorii logici** fac parte dintre operațiile din familia $(f_i)_{i \in I}$, adică există $i_1, i_2, i_3, i_4, i_5 \in I$, astfel încât:

  $n_{i_1} = 1$, $dom(f_{i_1}) = \{\text{true}, \text{false}\}$ și $\begin{cases} f_{i_1}(\text{true}) = \text{false și} \\ f_{i_1}(\text{false}) = \text{true}; \end{cases}$

  $n_{i_2} = n_{i_3} = n_{i_4} = n_{i_5} = 2$,

  $dom(f_2) = dom(f_3) = dom(f_4) = dom(f_5) = \{\text{true}, \text{false}\}^2$ și:

  $\begin{cases} f_{i_2}(\text{true}, \text{false}) = \text{false și} \\ f_{i_2}(x, y) = \text{true}, \text{ dacă } (x, y) \in \{\text{true}, \text{false}\}^2 \setminus \{\text{true}, \text{false}\}; \end{cases}$

  pentru orice $(x, y) \in \{\text{true}, \text{false}\}^2$, $f_{i_3} = f_{i_2}(f_{i_1}(x), y)$;

  pentru orice $(x, y) \in \{\text{true}, \text{false}\}^2$, $f_{i_4} = f_{i_1}(f_{i_2}(x, f_{i_1}(y)))$;

  pentru orice $(x, y) \in \{\text{true}, \text{false}\}^2$, $f_{i_5} = f_{i_4}(f_{i_2}(x, y), f_{i_2}(y, x))$.

Vom nota funcțiile de mai sus la fel ca pe conectorii logici cărora le corespund: $f_{i_1} = \neg$, $f_{i_2} = \rightarrow$, $f_{i_3} = \vee$, $f_{i_4} = \wedge$ și $f_{i_5} = \leftrightarrow$, cu aceeași regulă a priorităților pentru a evita parantezări excesive și cu scriere infixată pentru f_{i_2}, f_{i_3}, f_{i_4} și f_{i_5}, la fel ca în cazul conectorilor logici.

Pentru orice operație f cu $Im(f) \subseteq \{\text{true}, \text{false}\}$, compunerea $f_{i_1} \circ f$ va fi notată, simplu, $\neg f$. La fel pentru f_{i_2}, f_{i_3}, f_{i_4}, f_{i_5} aplicate unor termeni cu alți operatori dominanți.

Cu ce tipuri/cazuri particulare de formule lucrează Prologul?

De exemplu, Prologul acceptă următoarea bază de cunoștințe:

```
b(1).        % corespunde formulei fără variabile: b(1)

a(0).        % corespunde formulei fără variabile: a(0)
a(false).    % corespunde formulei fără variabile: a(false)
a(b(_)).     % corespunde formulei: ∀X a(b(X))
```

Dar la interogarea:

```
?- a(not(b(1))).    % corespunde formulei fără variabile: a(¬(b(1)))
```

răspunsul este **false**, deși $not(b(1))$ este evaluat la **false**, adică operația (mai precis compunerea de operații) $\neg\, b$ ia în argumentul 1 valoarea **false**;
răspunsul la interogarea de mai sus este **false** pentru că termenul $not(b(1))$, de operator dominant *not* (adică $\neg$ cu notația de mai sus), nu unifică, cu niciunul dintre termenii 0, *false* și $b(X)$, unde $X \in Var$ (termenii care apar ca argumente ale lui a în faptele de mai sus care definesc predicatul a).
În schimb, dacă adăugăm la baza de cunoștințe regula:

```
c(X) :- not(X).    % corespunde formulei care poate fi scrisă astfel pentru a
                   % concorda cu teoria anterioară: ∀X(¬(X=true) → c(X))
```

atunci la interogarea:

```
?- c(not(b(1))).    % corespunde formulei fără variabile: c(¬(b(1)))
```

Prologul răspunde **true**, pentru că X și $not(b(1))$ unifică, iar $b(1)$ e satisfăcut, prin urmare $not(b(1))$ nu e satisfăcut, așadar $not(not((b(1)))$ (adică $not(X)$ pentru X luând valoarea $not(b(1)))$ este satisfăcut, prin urmare $c(not(b(1)))$ (adică $c(X)$ pentru X luând valoarea $not(b(1)))$ e satisfăcut.
Observăm că Prologul nu acceptă fapte sau membri stângi de reguli conținând conectori logici. De exemplu, nu acceptă regula sau fapta următoare:

```
a(X), b(X) :- a(b(X)).
a(X); b(X).
```

În schimb, acceptă fapte sau reguli de forma:

```
d(a(X), b(X)).              % corespunde formulei: ∀X d(a(X) ∧ b(X))
e(a(X); b(X)) :- a(b(X)).% corespunde formulei: ∀X [a(b(X)) → e(a(X) ∨ b(X))]
```

Cu notațiile de operații de mai sus, argumentele predicatelor (relațiilor) d, respectiv e sunt **termenii** $a(X) \wedge b(X)$, respectiv $a(X) \vee b(X)$. A se vedea și fișierul *testfaptemsreguli.pl*; vom reveni la modul în care Prologul răspunde interogărilor. Așadar:

Cu ce tipuri/cazuri particulare de formule lucrează Prologul?

Faptele corespund unor formule de tipul:

$|\overline{\forall x_1 \forall x_2 \ldots \forall x_n \varphi}|$, unde $n \in \mathbb{N}^*$ și:

- φ este o formulă atomică;
- $V(\varphi) = FV(\varphi) = \{x_1, x_2, \ldots, x_n\}$, așadar $\forall x_1 \forall x_2 \ldots \forall x_n \varphi$ este un enunț.

Regulile corespund unor formule de tipul:

$|\overline{\forall x_1 \forall x_2 \ldots \forall x_n (\varphi \to \psi)}|$, unde $n \in \mathbb{N}^*$ și:

- φ este o formulă fără cuantificatori (φ este membrul drept al regulii);
- ψ este o formulă atomică (ψ este membrul stâng al regulii);
- așadar $V(\varphi \to \psi) = V(\varphi) \cup V(\psi) = FV(\varphi) \cup FV(\psi) = FV(\varphi \to \psi)$, deci $\varphi \to \psi$ este o formulă fără cuantificatori;
- $V(\varphi \to \psi) = \{x_1, x_2, \ldots, x_n\}$, așadar $\forall x_1 \forall x_2 \ldots \forall x_n (\varphi \to \psi)$ este un enunț.

Interogările corespund unor formule de tipul:

$|\overline{\exists x_1 \exists x_2 \ldots \exists x_n \varphi}|$, unde $n \in \mathbb{N}^*$ și:

- φ este o formulă fără cuantificatori;
- $V(\varphi) = FV(\varphi) = \{x_1, x_2, \ldots, x_n\}$, așadar $\exists x_1 \exists x_2 \ldots \exists x_n \varphi$ este un enunț.

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Axiomele logicii clasice a predicatelor

Axiomele calculului cu predicate clasic: pentru φ, ψ, χ formule arbitrare, t termen arbitrar, n, i numere naturale nenule arbitrare și $x, y, y_1, \ldots, y_n, v_1, \ldots, v_n$ variabile arbitrare:

- axiomele calculului propozițional:
  (G_1) $\varphi \to (\psi \to \varphi)$
  (G_2) $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
  (G_3) $(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$
- *regula* $(\to \forall)$:
  (G_4) $\forall x(\varphi \to \psi) \to (\varphi \to \forall x\psi)$, dacă $x \notin FV(\varphi)$
- o regulă privind substituțiile:
  (G_5) $\forall x\varphi(x, y_1, \ldots, y_n) \to \varphi(t, y_1, \ldots, y_n)$
- *axiomele egalității*:
  (G_6) $x{=}x$
  (G_7)
  $(x{=}y) \to (t(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n){=}t(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_n))$
  (G_8)
  $(x{=}y) \to (\varphi(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n){=}\varphi(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_n))$

Teoremele formale, i.e. adevărurile sintactice în logica clasică a predicatelor

Notație

Faptul că o formulă φ este *teoremă (formală) (adevăr sintactic) a(l) lui* $\mathcal{L}_\tau$ se notează cu $\vdash \varphi$ și se definește, recursiv, ca mai jos.

Definiție

1. Orice axiomă e teoremă formală a lui $\mathcal{L}_\tau$.
2. Pentru orice formule φ, ψ, $\dfrac{\vdash \psi, \vdash \psi \to \varphi}{\vdash \varphi}$ (regula de deducție *modus ponens* (MP)).
3. Pentru orice formulă φ și orice variabilă x, $\dfrac{\vdash \varphi}{\vdash \forall x \varphi}$ (regula de deducție numită *principiul generalizării* (PG)).
4. Orice teoremă formală se obține prin aplicarea regulilor (1), (2) și (3) de un număr finit de ori.

Deducție sintactică în logica clasică a predicatelor

Notație

Fie Σ o mulțime de formule ale lui $\mathcal{L}_\tau$. Faptul că o formulă φ *se deduce (formal) din ipotezele* Σ (φ *este consecință sintactică a mulțimii de ipoteze* Σ) se notează cu $\Sigma \vdash \varphi$ și se definește, recursiv, ca mai jos.

Definiție

Fie Σ o mulțime de formule ale lui $\mathcal{L}_\tau$.

1. Orice axiomă a lui $\mathcal{L}_\tau$ se deduce formal din Σ.
2. $\Sigma \vdash \varphi$, oricare ar fi $\varphi \in \Sigma$.
3. Pentru orice formule φ, ψ, $\dfrac{\Sigma \vdash \psi, \Sigma \vdash \psi \to \varphi}{\Sigma \vdash \varphi}$ (regula de deducție *modus ponens* (MP)).
4. Pentru orice formulă φ și orice variabilă x, $\dfrac{\Sigma \vdash \varphi}{\Sigma \vdash \forall x \varphi}$ (regula de deducție numită *principiul generalizării* (PG)).
5. Orice consecință sintactică a lui Σ se obține prin aplicarea regulilor (1), (2), (3) și (4) de un număr finit de ori.

Teorema deducției sintactice în logica clasică a predicatelor

Remarcă

Pentru orice formulă φ, are loc echivalența:

$$\emptyset \vdash \varphi \Leftrightarrow \vdash \varphi.$$

Teoremă (Teorema deducției)

Pentru orice mulțime de **formule** Σ, *orice* **enunț** φ *și orice* **formulă** ψ, *are loc echivalența:*

$$\Sigma \vdash \varphi \to \psi \Leftrightarrow \Sigma \cup \{\varphi\} \vdash \psi.$$

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Fie:

- $\tau = ((n_i)_{i\in I}; (m_j)_{j\in J}; (0)_{k\in K})$ o signatură;
- $\mathcal{A} = (A; (f_i)_{i\in I}; (R_j)_{j\in J}; (c_k)_{k\in K})$ o structură de ordinul I de signatură τ;

  notăm: $\begin{cases} \mathcal{F} = \{f_i \mid i \in I\}, \\ \mathcal{R} = \{R_j \mid j \in J\}, \\ \mathcal{C} = \{c_k \mid k \in K\}; \end{cases}$ adică:

  $\mathcal{F}$ este mulțimea operațiilor cu argumente ale lui $\mathcal{A}$;
  $\mathcal{R}$ este mulțimea relațiilor lui $\mathcal{A}$;
  $\mathcal{C}$ este mulțimea constantelor lui $\mathcal{A}$.

Amintesc că am notat cu *Var* mulțimea **variabilelor** din limbajul $\mathcal{L}_\tau$ (limbajul de ordinul I asociat signaturii τ).

Elementele lui *Var* nu sunt variabile propoziționale, ci sunt variabilele care apar în predicate, i.e. în propozițiile cu variabile.

În continuare ne vom referi la **termenii** și **formulele** din limbajul $\mathcal{L}_\tau$.

Definiție

O *interpretare* (sau *evaluare*, sau *semantică*) a limbajului $\mathcal{L}_\tau$ în structura algebrică $\mathcal{A}$ este o funcție $s : Var \to A$.
Fiecare variabilă $x \in Var$ este "interpretată" prin elementul $s(x) \in A$.

Prelungim o interpretare s la mulțimea tuturor termenilor, obținând o funcție $s : Term(\mathcal{L}_\tau) \to A$.

Definiție (valorile asociate termenilor de o interpretare)

Pentru orice interpretare s și orice termen t, definim recursiv elementul $s(t) \in A$:

- dacă $t = x \in Var$, atunci $s(t) = s(x)$;
- dacă $t = c \in C$, atunci $s(t) = c$;
- dacă $t = f(t_1, \ldots, t_n)$, unde $f \in \mathcal{F}$ are aritatea $n \in \mathbb{N}^*$, iar $t_1, \ldots, t_n \in Term(\mathcal{L}_\tau)$, atunci $s(t) = f(s(t_1), \ldots, s(t_n))$.

Notație

Pentru orice interpretare $s : Var \to A$, orice $x \in Var$ și orice $a \in A$, notăm cu $s[^x_a] : Var \to A$ interpretarea definită prin: oricare ar fi $v \in Var$,

$$s[^x_a](v) = \begin{cases} a, & \text{dacă } v = x, \\ s(v), & \text{dacă } v \neq x. \end{cases}$$

$s[^x_a]$ ia valoarea a în variabila x și aceeași valoare ca s în celelalte variabile.

Acum definim valorile unei interpretări s în formule; acestea vor fi valori din **algebra Boole standard**, **algebra Boole a valorilor de adevăr** $\mathcal{L}_2 = (\{0,1\}, \vee, \wedge, \bar{\cdot}, 0, 1)$, unde $0 \neq 1$, $\vee$ este **disjuncția** și $\wedge$ este **conjuncția** (operații de latice), $\bar{\cdot}$ este **complementul** (operație booleană), 0 este **primul element**, iar 1 este **ultimul element**. Notăm cu $\rightarrow$ **implicația booleană**, iar cu $\leftrightarrow$ **echivalența booleană** în $\mathcal{L}_2$ (operații booleene derivate). A se revedea lecția de curs recapitulativă despre algebrele Boole.

Astfel, obținem o funcție $s : Term(\mathcal{L}_\tau) \cup Form(\mathcal{L}_\tau) \rightarrow A \cup \mathcal{L}_2$.

Definiție (valorile booleene asociate formulelor de o interpretare)

Pentru orice interpretare s și orice formulă φ, *valoarea de adevăr a lui φ în interpretarea s* este un element din algebra Boole standard $\mathcal{L}_2 = \{0,1\}$, notat cu $s(\varphi)$, definit, recursiv, astfel:

- dacă $\varphi = (t_1 \mathbf{=} t_2)$, pentru doi termeni t_1, t_2, atunci
$$s(\varphi) = \begin{cases} 1, & \text{dacă } s(t_1) = s(t_2), \\ 0, & \text{dacă } s(t_1) \neq s(t_2) \end{cases}$$
- dacă $\varphi = R(t_1, \ldots, t_m)$, unde $R \in \mathcal{R}$ are aritatea $m \in \mathbb{N}^*$, iar $t_1, \ldots, t_m$ sunt termeni, atunci $s(\varphi) = \begin{cases} 1, & \text{dacă } (s(t_1), \ldots, s(t_m)) \in R, \\ 0, & \text{dacă } (s(t_1), \ldots, s(t_m)) \notin R \end{cases}$

Definiție (continuare – am definit mai sus valorile lui φ în formulele atomice; acum extindem definiția lui φ la toate formulele)

- dacă $\varphi = \neg\psi$, pentru o formulă ψ, atunci $s(\varphi) = \overline{s(\psi)}$;
- dacă $\varphi = \psi \to \chi$, pentru două formule ψ, χ, atunci $s(\varphi) = s(\psi) \to s(\chi)$;
- dacă $\varphi = \forall x\psi$, unde $x \in Var$, iar ψ este o formulă, atunci $s(\varphi) = \bigwedge_{a \in A} s[^x_a](\psi)$.

Remarcă

Este imediat că, pentru orice interpretare $s : Var \to A$, orice formule ψ, χ și orice $x \in Var$, au loc egalitățile:

- $s(\psi \vee \chi) = s(\psi) \vee s(\chi)$;
- $s(\psi \wedge \chi) = s(\psi) \wedge s(\chi)$;
- $s(\psi \leftrightarrow \chi) = s(\psi) \leftrightarrow s(\chi)$;
- $s(\exists x\psi) = \bigvee_{a \in A} s[^x_a](\psi)$.

Fie $s : Var \to A$, $x \in Var$ și $\psi \in Form(\mathcal{L}_\tau)$. Cum $s(\alpha) \in \mathcal{L}_2 = \{0, 1\}$ pentru orice $\alpha \in Form(\mathcal{L}_\tau)$, din cele de mai sus rezultă că: $s(\forall x\psi) = 1$ ddacă $s[^x_a](\psi) = 1$ pentru toți $a \in A$, iar $s(\exists x\psi) = 1$ ddacă există un $a \in A$ a.î. $s[^x_a](\psi) = 1$.

Lemă (dacă două interpretări coincid pe variabilele dintr–un termen, atunci ele coincid în acel termen)

Fie $s_1, s_2 : Var \to A$ două interpretări. Atunci, pentru orice termen t, are loc implicația: $s_1 \mid_{V(t)} = s_2 \mid_{V(t)} \Rightarrow s_1(t) = s_2(t)$.

Propoziție (dacă două interpretări coincid pe variabilele libere dintr–o formulă, atunci ele coincid în acea formulă)

Fie $s_1, s_2 : Var \to A$ două interpretări. Atunci, pentru orice formulă φ, are loc implicația: $s_1 \mid_{FV(\varphi)} = s_2 \mid_{FV(\varphi)} \Rightarrow s_1(\varphi) = s_2(\varphi)$.

În mod trivial, oricare două interpretări $s_1, s_2 : Var \to A$ coincid pe $\emptyset$: $(\forall x)(x \in \emptyset \Rightarrow s_1(x) = s_2(x))$ e adevărată, pentru că $x \in \emptyset$ e falsă pentru orice x, așadar $x \in \emptyset \Rightarrow s_1(x) = s_2(x)$ e adevărată pentru orice x. Prin urmare:

Corolar (cum enunțurile nu au variabile libere (i.e. $FV(\varphi) = \emptyset$ pt. orice enunț φ), rezultă că, într–un enunț, toate interpretările au aceeași valoare)

Dacă φ este un enunț, atunci $s(\varphi)$ nu depinde de interpretarea $s : Var \to A$.

Notație (această valoare nu depinde decât de structura algebrică $\mathcal{A}$)

Corolarul anterior ne permite să notăm, pentru orice enunț φ, cu $||\varphi||_{\mathcal{A}} = s(\varphi)$ valoarea oricărei interpretări $s : Var \to A$ în enunțul φ.

Satisfiabilitate, adevăruri semantice, deducție semantică

Definiție (modele pentru enunțuri și mulțimi de enunțuri)

Pentru orice enunț φ, notăm:

$$\mathcal{A} \models \varphi \text{ ddacă } ||\varphi||_{\mathcal{A}} = 1,$$

și, în acest caz, spunem că $\mathcal{A}$ *satisface* φ sau φ *este adevărat în* $\mathcal{A}$ sau $\mathcal{A}$ *este model pentru* φ.
Pentru orice mulțime Γ de enunțuri, notăm:

$$\mathcal{A} \models \Gamma \text{ ddacă } (\forall\, \gamma \in \Gamma)\,(\mathcal{A} \models \gamma),$$

și, în acest caz, spunem că $\mathcal{A}$ *satisface* Γ sau că $\mathcal{A}$ *este model pentru* Γ.

Definiție (satisfiabilitate)

Spunem că un enunț φ *e satisfiabil* sau *are un model* ddacă există o structură de ordinul I $\mathcal{M}$ de signatură τ astfel încât $\mathcal{M} \models \varphi$.
Spunem că o mulțime Γ de enunțuri *e satisfiabilă* sau *are un model* ddacă există o structură de ordinul I $\mathcal{M}$ de signatură τ astfel încât $\mathcal{M} \models \Gamma$.

Definiție (adevărurile semantice și deducția semantică)

Fie φ un enunț. Spunem că φ este *universal adevărat (adevăr semantic, tautologie)* ddacă $\mathcal{M} \models \varphi$ pentru orice structură de ordinul I $\mathcal{M}$ de signatură τ.

Echivalența semantică pe mulțimea enunțurilor lui $\mathcal{L}_\tau$

Definiție (continuare)

Notăm acest lucru cu $\models \varphi$.
Fie Γ o mulțime de enunțuri. Spunem că φ este *consecință semantică* a lui Γ sau φ enunț *deductibil semantic din* Γ ddacă orice model pentru Γ este model pentru φ, adică, pentru orice structură de ordinul I $\mathcal{M}$ de signatură τ, $\mathcal{M} \models \Gamma$ implică $\mathcal{M} \models \varphi$. Notăm acest lucru cu $\Gamma \models \varphi$.

Amintesc că mulțimea enunțurilor lui $\mathcal{L}_\tau$ este $\{\varphi \in Form(\mathcal{L}_\tau) \mid FV(\varphi) = \emptyset\}$.

Definiție (echivalența semantică)

Considerăm pe pe mulțimea enunțurilor din limbajul $\mathcal{L}_\tau$ o relație binară notată $\models\!\!\!\dashv$, definită astfel: oricare ar fi enunțurile φ și ψ, $\varphi \models\!\!\!\dashv \psi$ ddacă, oricare ar fi structura algebrică $\mathcal{M}$ de signatură τ, avem: $\mathcal{M} \models \varphi$ ddacă $\mathcal{M} \models \psi$.
Adică o pereche de enunțuri $(\varphi, \psi) \in \models\!\!\!\dashv$ ddacă φ și ψ au aceleași modele.
Relația binară $\models\!\!\!\dashv$ se numește *echivalența semantică* în limbajul $\mathcal{L}_\tau$. Pentru orice enunțuri φ, ψ, spunem că φ și ψ sunt *echivalente semantic* ddacă $(\varphi, \psi) \in \models\!\!\!\dashv$.

Remarcă

$\models\!\!\!\dashv$ este o relație de echivalență pe mulțimea enunțurilor lui $\mathcal{L}_\tau$. Într–adevăr, reflexivitatea, simetria și tranzitivitatea lui $\models\!\!\!\dashv$ sunt imediate.

Remarcă (două enunțuri sunt echivalente semantic ddacă echivalența lor logică este tautologie)

Pentru orice enunțuri φ, ψ, avem:

$$\varphi \models\!\!\!\mid \psi \text{ ddacă } \models \varphi \leftrightarrow \psi.$$

Într–adevăr, în primul rând să observăm că $\varphi \leftrightarrow \psi$ este un enunț, întrucât $FV(\varphi \leftrightarrow \psi) = FV(\varphi) \cup FV(\psi) = \emptyset \cup \emptyset = \emptyset$.
Acum, considerând o interpretare arbitrară $s : Var \to A$, avem:
$||\varphi \leftrightarrow \psi||_{\mathcal{A}} = s(\varphi \leftrightarrow \psi) = s(\varphi) \leftrightarrow s(\psi) = ||\varphi||_{\mathcal{A}} \leftrightarrow ||\psi||_{\mathcal{A}}$.
Prin urmare: $\models \varphi \leftrightarrow \psi$ ddacă, pentru orice algebră $\mathcal{M}$ de tip τ, $\mathcal{M} \models \varphi \leftrightarrow \psi$, dacă, pentru orice algebră $\mathcal{M}$ de tip τ, $||\varphi \leftrightarrow \psi||_{\mathcal{M}} = 1$, dacă, pentru orice algebră $\mathcal{M}$ de tip τ, $||\varphi||_{\mathcal{M}} \leftrightarrow ||\psi||_{\mathcal{M}} = 1$, dacă, pentru orice algebră $\mathcal{M}$ de tip τ, $||\varphi||_{\mathcal{M}} = ||\psi||_{\mathcal{M}}$, dacă, pentru orice algebră $\mathcal{M}$ de tip τ, avem echivalența: $||\varphi||_{\mathcal{M}} = 1 \Leftrightarrow ||\psi||_{\mathcal{M}} = 1$, dacă, pentru orice algebră $\mathcal{M}$ de tip τ, avem echivalența: $\mathcal{M} \models \varphi \Leftrightarrow \mathcal{M} \models \psi$, ddacă $\varphi \models\!\!\!\mid \psi$.
Am folosit următoarele fapte:

- pentru orice elemente a, b ale unei algebre Boole, avem: $a \leftrightarrow b = 1$ ddacă $a = b$;
- oricare ar fi algebra $\mathcal{M}$ de tip τ: $||\varphi||_{\mathcal{M}}, ||\psi||_{\mathcal{M}} \in \mathcal{L}_2 = \{0, 1\}$, așadar $||\varphi||_{\mathcal{M}} = ||\psi||_{\mathcal{M}}$ ddacă fie ambele sunt egale cu 1, fie ambele sunt egale cu 0.

Remarcă

Conform remarcii anterioare, pentru orice enunțuri φ, ψ, avem: $\varphi \models\!\!\!\dashv \psi$ ddacă, pentru orice algebră $\mathcal{M}$ de tip τ, $||\varphi||_{\mathcal{M}} = ||\psi||_{\mathcal{M}}$.
Din această exprimare a relației $\models\!\!\!\dashv$ de echivalență semantică pentru $\mathcal{L}_\tau$ rezultă că $\models\!\!\!\dashv$ are proprietățile relației $\models\!\!\!\dashv$ de echivalență semantică din logica propozițională clasică, anume cele provenite din proprietățile booleene *(a se vedea mai jos)*.
În plus, au loc proprietăți de tipul următor *(a se vedea o listă extinsă mai jos)*, pentru orice $x, y \in Var$ și orice $\alpha, \beta \in Form(\mathcal{L}_\tau)$:

- dacă $FV(\alpha) \subseteq \{x, y\}$, astfel că următoarele formule sunt enunțuri, atunci: $\forall x \forall y \alpha \ \models\!\!\!\dashv \ \forall y \forall x \alpha$;
- dacă α este un enunț, $x \notin V(\alpha)$ și $FV(\beta) \subseteq \{x\}$, astfel că următoarele formule sunt enunțuri, atunci: $\alpha \ \models\!\!\!\dashv \ \forall x \alpha \ \models\!\!\!\dashv \ \exists x \alpha$ și $\forall x(\alpha \vee \beta) \ \models\!\!\!\dashv \ \alpha \vee \forall x \beta \ \models\!\!\!\dashv \ \forall x \alpha \vee \forall x \beta$.

Într–adevăr, considerând $x, y \in Var$, două **formule arbitrare** α, β, o algebră $\mathcal{M}$ de tip τ, cu mulțimea elementelor M, și o interpretare $s : Var \to M$, avem: notând, pentru orice $a, b \in M$, cu $t_{a,b} : Var \to M$ interpretarea definită prin:

oricare ar fi $v \in Var$, $t_{a,b}(v) = \begin{cases} a, & \text{dacă } v = x, \\ b, & \text{dacă } v = y, \\ s(v), & \text{dacă } v \notin \{x, y\}, \end{cases}$ observăm că

$t_{a,b} = (s[^x_a])[^y_b] = (s[^y_b])[^x_a]$, așadar:

$$s(\forall x \forall y \alpha) = \bigwedge_{a \in A} s[^x_a](\forall y \alpha) = \bigwedge_{a \in A} \bigwedge_{b \in A} (s[^x_a])[^y_b](\alpha) = \bigwedge_{a \in A} \bigwedge_{b \in A} t_{a,b}(\alpha) =$$

$$\bigwedge_{b \in A} \bigwedge_{a \in A} t_{a,b}(\alpha) = \bigwedge_{b \in A} \bigwedge_{a \in A} (s[^y_b])[^x_a](\alpha) = \bigwedge_{b \in A} s[^y_b](\forall x \alpha) = s(\forall y \forall x \alpha);$$

iar, dacă $\forall x \forall y \alpha$ este un **enunț**, adică $FV(\alpha) \subseteq \{x, y\}$, astfel că și $\forall y \forall x \alpha$ este un enunț, atunci, conform calculului anterior,
$||\forall x \forall y \alpha||_{\mathcal{M}} = s(\forall x \forall y \alpha) = s(\forall y \forall x \alpha) = ||\forall x \forall y \alpha||_{\mathcal{M}}$, **și la fel mai jos**;

- dacă $x \notin FV(\alpha)$, în particular dacă $x \notin V(\alpha)$, atunci, pentru orice $a \in A$, $s\mid_{FV(\alpha)} = s[^x_a]\mid_{FV(\alpha)}$, așadar, conform propoziției precedente, pentru orice $a \in A$, $s(\alpha) = s[^x_a](\alpha)$, prin urmare:

$$s(\forall x \alpha) = \bigwedge_{a \in A} s[^x_a](\alpha) = s(\alpha) = \bigvee_{a \in A} s[^x_a](\alpha) = s(\exists x \alpha);$$

întrucât algebra Boole $\mathcal{L}_2$ este finită și, în particular, completă, fapt pe care l–am folosit deja când am admis scrieri de tipul $\bigwedge_{a \in A} s[^x_a](\alpha)$ sau $\bigvee_{a \in A} s[^x_a](\alpha)$, deci am folosit existența conjuncțiilor și disjuncțiilor arbitrare, avem:

$$s(\forall x(\alpha \vee \beta)) = \bigwedge_{a \in A} s[^x_a](\alpha \vee \beta) = \bigwedge_{a \in A} (s[^x_a](\alpha) \vee s[^x_a](\beta)) = \bigwedge_{a \in A} (s(\alpha) \vee s[^x_a](\beta)) =$$

$s(\alpha) \vee \bigwedge_{a \in A} s[^x_a](\beta) = s(\alpha) \vee s(\forall x \beta) = s(\alpha \vee \forall x \beta)$, dar, conform celor de mai sus, avem și: $s(\alpha) \vee s(\forall x \beta) = s(\forall x \alpha) \vee s(\forall x \beta) = s(\forall x \alpha \vee \forall x \beta)$.

Să ne amintim ce sunt substituțiile

Definiție (substituțiile sunt funcțiile de la variabile la termeni)

O *substituție* este o funcție $\sigma : Var \to Term(\mathcal{L}_\tau)$.

Definiție

Fie $\sigma : Var \to Term(\mathcal{L}_\tau)$. Următoarea extindere a lui σ la $Term(\mathcal{L}_\tau)$ se numește tot *substituție* (și, de obicei, se notează tot cu σ): $\widetilde{\sigma} : Term(\mathcal{L}_\tau) \to Term(\mathcal{L}_\tau)$, definită, recursiv, astfel:

- $\widetilde{\sigma}\mid_{Var} = \sigma$;
- oricare ar fi $c \in \mathcal{C}$, $\widetilde{\sigma}(c) = c$;
- pentru orice $n \in \mathbb{N}^*$, orice $f \in \mathcal{F}$ având aritatea n și orice $t_1, \ldots, t_{n_i} \in Term(\mathcal{L}_\tau)$, $\widetilde{\sigma}(f(t_1, \ldots, t_{n_i})) = f(\widetilde{\sigma}(t_1), \ldots, \widetilde{\sigma}(t_n))$.

Observație

Extinderea $\widetilde{\sigma}$ din definiția anterioară este:

- **complet definită**, pentru că toți termenii se scriu ca mai sus, i.e. sunt obținuți prin acea recursie;
- **corect definită**, pentru că orice termen are o unică scriere ca mai sus.

Definiție (definim substituțiile pe formulele atomice, apoi, recursiv, pe toate formulele fără cuantificatori)

Fie $\sigma : Term(\mathcal{L}_\tau) \to Term(\mathcal{L}_\tau)$ o substituție (adică o extindere ca în definiția anterioară a unei funcții de la Var la $Term(\mathcal{L}_\tau)$).
Definim σ pe formulele atomice, recursiv, astfel:

- pentru orice $t_1, t_2 \in Term(\mathcal{L}_\tau)$, $\sigma(t_1{=}t_2) \ = \ \sigma(t_1){=}\sigma(t_2)$;
- pentru orice $m \in \mathbb{N}^*$, orice $R \in \mathcal{R}$ de aritate m și orice $t_1, \ldots, t_m \in Term(\mathcal{L}_\tau)$, $\sigma(R(t_1, \ldots, t_m)) = R(\sigma(t_1), \ldots, \sigma(t_m))$.

Practic, mai sus, obținem valorile lui σ în termenii obținuți din formulele atomice $t_1{=}t_2$, respectiv $R(t_1, \ldots, t_m)$ prin înlocuirea relațiilor din acești termeni cu operațiile de rezultat boolean asociate acestora, care, într–o algebră $\mathcal{A}$ de signatură τ, având mulțimea elementelor $A \supseteq \mathcal{L}_2 = \{0, 1\}$, sunt definite prin:

- $=^{\mathcal{A}} : A^2 \to \mathcal{L}_2 \subseteq A$, pentru orice $a, b \in A$, $=^{\mathcal{A}}(a, b) = \begin{cases} 1, & \text{dacă } a = b, \\ 0, & \text{dacă } a \neq b; \end{cases}$
- $R^{\mathcal{A}} : A^m \to \mathcal{L}_2 \subseteq A$, pentru orice $a_1, \ldots, a_m \in A$,
  $R^{\mathcal{A}}(a_1, \ldots, a_m) = \begin{cases} 1, & \text{dacă } (a_1, \ldots, a_m) \in R^{\mathcal{A}}, \\ 0, & \text{dacă } (a_1, \ldots, a_m) \notin R^{\mathcal{A}}. \end{cases}$

Definiție (continuare – definiția recursivă a unei substituții pe formulele non–atomice fără cuantificatori, adică formulele cu conectori logici și fără cuantificatori)

După ce obținem valoarea lui σ într–o formulă atomică α asociind, ca mai sus, un termen lui α, facem operația inversă pentru termenul $\sigma(\alpha)$: înlocuim operația dominantă (având rezultatul boolean) a acestui termen cu relația căreia i–am asociat această operație, și astfel termenul $\sigma(\alpha)$ devine formulă atomică.

Pentru orice formule fără cuantificatori φ, ψ, definim:

- $\sigma(\neg\varphi) = \neg\sigma(\varphi)$;
- $\sigma(\varphi \to \psi) = \sigma(\varphi) \to \sigma(\psi)$.

Din definiția anterioară rezultă:

Remarcă (pentru conectorii logici derivați)

Pentru orice $\sigma : \mathit{Term}(\mathcal{L}_\tau) \to \mathit{Term}(\mathcal{L}_\tau)$ și orice formule fără cuantificatori φ, ψ, au loc:

- $\sigma(\varphi \vee \psi) = \sigma(\varphi) \vee \sigma(\psi)$;
- $\sigma(\varphi \wedge \psi) = \sigma(\varphi) \wedge \sigma(\psi)$;
- $\sigma(\varphi \leftrightarrow \psi) = \sigma(\varphi) \leftrightarrow \sigma(\psi)$.

Ce definește un program în Prolog și ce semnifică interogările?

Considerăm o signatură τ, un $n \in \mathbb{N}^*$ și o bază de cunoștințe în Prolog formată din n fapte și reguli corespunzătoare formulelor $\varphi_1, \ldots, \varphi_n \in Form(\mathcal{L}_\tau)$.

Această bază de cunoștințe definește clasa tuturor algebrelor de signatură τ care satisfac mulțimea de enunțuri $\{\varphi_1, \ldots, \varphi_n\}$, adică mulțimea modelelor lui $\{\varphi_1, \ldots, \varphi_n\}$.

Dacă o interogare corespunde formulei $\varphi = \exists\, x_1\, \exists\, x_2\, \ldots \exists\, x_k\, \psi \in Form(\mathcal{L}_\tau)$, unde ψ este o formulă fără cuantificatori, atunci a rezolva interogarea respectivă semnifică a determina dacă are loc deducția semantică:

$$\{\varphi_1, \ldots, \varphi_n\} \models \varphi,$$

adică dacă orice algebră de signatură τ care satisface mulțimea de enunțuri $\{\varphi_1, \ldots, \varphi_n\}$ satisface și pe φ, adică dacă orice algebră din clasa algebrelor definite de această bază de cunoștințe este model pentru φ.
Mai precis, după cum vom vedea, a rezolva această interogare înseamnă a găsi substituțiile $\sigma = \{x_1/t_1, \ldots, x_k/t_k\}$, unde $t_1, \ldots, t_k$ sunt termeni având $V(t_1) \cup \ldots \cup V(t_k) = \{v_1, \ldots, v_j\}$, cu proprietatea că:

$$\{\varphi_1, \ldots, \varphi_n\} \models \forall\, v_1\, \forall\, v_2\, \ldots \forall\, v_j\, \sigma(\psi).$$

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Formă prenex și formă normală conjunctivă prenex

Fie τ și $\mathcal{A}$ ca în secțiunea anterioară a cursului.

Definiție

Se numește *formulă prenex* sau *formulă în formă prenex* o formulă de tipul

$$\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n \varphi,$$

unde $n \in \mathbb{N}^*$, $x_1, \ldots, x_n \in Var$, $\mathcal{Q}_1, \ldots, \mathcal{Q}_n \in \{\forall, \exists\}$ și φ este o formulă fără cuantificatori.

Observație

Toate formulele cu care lucrează Prologul sunt în formă prenex: atât cele corespunzătoare **faptelor** și **regulilor**, cât și cele corespunzătoare **interogărilor**.

Remarcă (orice enunț poate fi pus în formă prenex)

Pentru orice enunț ε, există o formulă prenex ψ astfel încât $\varepsilon \models\!\!\!\dashv \psi$.

Putem obține o formulă prenex echivalentă semantic cu ε folosind următoarele echivalențe semantice valabile pentru orice enunțuri α, β, γ, orice formule φ, ψ, χ și orice variabile x, y a.î. $FV(\varphi) \cup FV(\psi) \subseteq \{x\}$, iar $FV(\chi) \subseteq \{x, y\}$, astfel că următoarele formule sunt enunțuri:

(proprietăți pentru mutarea cuantificatorilor în față)

- $\forall x \forall y \chi \models\!\!\!\dashv \forall y \forall x \chi$ și $\exists x \exists y \chi \models\!\!\!\dashv \exists y \exists x \chi$
- $\neg \forall x \chi \models\!\!\!\dashv \exists x \neg \chi$ și $\neg \exists x \chi \models\!\!\!\dashv \forall x \neg \chi$
- $\forall x(\varphi \wedge \psi) \models\!\!\!\dashv \forall x \varphi \wedge \forall x \psi$ și $\exists x(\varphi \vee \psi) \models\!\!\!\dashv \exists x \varphi \vee \exists x \psi$
- dacă $x \notin V(\varphi)$, atunci: $\begin{cases} \varphi \models\!\!\!\dashv \forall x \varphi \models\!\!\!\dashv \exists x \varphi \\ \varphi \vee \forall x \psi \models\!\!\!\dashv \forall x(\varphi \vee \psi) \models\!\!\!\dashv \forall x \varphi \vee \forall x \psi \\ \varphi \wedge \exists x \psi \models\!\!\!\dashv \exists x(\varphi \wedge \psi) \models\!\!\!\dashv \exists x \varphi \wedge \exists x \psi \end{cases}$

(proprietăți din calculul propozițional)

1. $\alpha \to \beta \models\!\!\!\dashv \neg \alpha \vee \beta$ și $\alpha \leftrightarrow \beta \models\!\!\!\dashv (\neg \alpha \vee \beta) \wedge (\neg \beta \vee \alpha)$
2. $\alpha \vee \alpha \models\!\!\!\dashv \alpha \wedge \alpha \models\!\!\!\dashv \alpha$
3. $\alpha \vee \beta \models\!\!\!\dashv \beta \vee \alpha$ și $\alpha \wedge \beta \models\!\!\!\dashv \beta \wedge \alpha$
4. $(\alpha \vee \beta) \vee \gamma \models\!\!\!\dashv \alpha \vee (\beta \vee \gamma)$ și $(\alpha \wedge \beta) \wedge \gamma \models\!\!\!\dashv \alpha \wedge (\beta \wedge \gamma)$
5. $\alpha \vee (\alpha \wedge \beta) \models\!\!\!\dashv \alpha \wedge (\alpha \vee \beta) \models\!\!\!\dashv \alpha$
6. $\alpha \vee (\beta \wedge \gamma) \models\!\!\!\dashv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$ și $\alpha \wedge (\beta \vee \gamma) \models\!\!\!\dashv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
7. $\neg \neg \alpha \models\!\!\!\dashv \alpha$
8. $\neg(\alpha \vee \beta) \models\!\!\!\dashv \neg \alpha \wedge \neg \beta$ și $\neg(\alpha \wedge \beta) \models\!\!\!\dashv \neg \alpha \vee \neg \beta$
9. $\alpha \vee (\beta \wedge \neg \beta) \models\!\!\!\dashv \alpha \vee (\beta \wedge \neg \beta \wedge \gamma) \models\!\!\!\dashv \alpha \wedge (\beta \vee \neg \beta) \models\!\!\!\dashv \alpha \wedge (\beta \vee \neg \beta \vee \gamma) \models\!\!\!\dashv \alpha$

Remarcă

Cum, în ultima remarcă din secțiunea anterioară a cursului, α și β sunt **formule arbitrare**, rezultă că **echivalențele semantice** de mai sus au loc și în **subformule** care nu sunt neapărat enunțuri, adică putem aplica faptul că, pentru orice $\varphi, \psi \in Form(\mathcal{L}_\tau)$, orice algebră $\mathcal{M}$ de signatură τ, cu mulțimea elementelor M, și orice interpretare $s : Var \to M$, au loc, de exemplu:

- $s(\neg(\alpha \vee \beta)) = s(\neg\alpha \wedge \neg\beta)$, $s(\exists x \exists y \alpha) = s(\exists y \exists x \alpha)$;
- dacă $x \notin FV(\alpha)$, atunci $s(\alpha) = s(\forall x \alpha) = s(\exists x \alpha)$ și $s(\alpha \wedge \exists x \beta) = s(\exists x(\alpha \wedge \beta)) = s(\exists x \alpha \wedge \exists x \beta)$ etc..

Definiție (FNC prenex)

- Un *literal* este o formulă atomică sau negația unei formule atomice.
- O *clauză* este o disjuncție de literali.

Orice clauză se identifică cu mulțimea literalilor care o compun.

- O *formă normală conjunctivă prenex* sau un *enunț în formă normală conjunctivă prenex* (*FNC prenex*) este un enunț de forma:

$$\forall x_1 \ldots \forall x_n \varphi,$$

unde $n \in \mathbb{N}^*$, $x_1, \ldots, x_n \in Var$ și φ este o conjuncție de clauze, i. e. o

Definiție (continuare)

conjuncție de disjuncții de literali, implicit φ este o formulă fără cuantificatori, așadar $x_1, \ldots, x_n \in Var$ este o formulă prenex.

Orice conjuncție de clauze se identifică, cu mulțimea acelor clauze.
Orice FNC prenex $\forall x_1 \ldots \forall x_n \varphi$ se identifică cu mulțimea clauzelor care îl compun pe φ.

Observație

Întrucât toate enunțurile au lungime finită, conjuncțiile și disjuncțiile la care face referire definiția de mai sus sunt finite.

Remarcă

O mulțime finită și nevidă de enunțuri este satisfiabilă ddacă, conjuncția enunțurilor din acea mulțime e satisfiabilă.
Într–adevăr, dacă $n \in \mathbb{N}^*$ și $\gamma_1, \ldots, \gamma_n$ sunt enunțuri, iar $\mathcal{M}$ este o algebră de signatură τ, atunci: $||\gamma_1 \wedge \ldots \wedge \gamma_n||_{\mathcal{M}} = ||\gamma_1||_{\mathcal{M}} \wedge \ldots \wedge ||\gamma_n||_{\mathcal{M}}$, așadar:
$\mathcal{M} \models \gamma_1 \wedge \ldots \wedge \gamma_n$ ddacă $||\gamma_1 \wedge \ldots \wedge \gamma_n||_{\mathcal{M}} = 1$ ddacă $||\gamma_1||_{\mathcal{M}} \wedge \ldots \wedge ||\gamma_n||_{\mathcal{M}} = 1$ ddacă $||\gamma_1||_{\mathcal{M}} = \ldots = ||\gamma_n||_{\mathcal{M}} = 1$ ddacă, pentru fiecare $i \in \overline{1, n}$, $\mathcal{M} \models \gamma_i$, ddacă $\mathcal{M} \models \{\gamma_1, \ldots, \gamma_n\}$.

Remarcă

În mod trivial, mulțimea vidă de enunțuri este satisfiabilă, întrucât următoarea afirmație este adevărată: pentru orice algebră $\mathcal{M}$ de tip τ și orice γ, $\gamma \in \emptyset \Rightarrow \mathcal{M} \models \gamma$.

Definiție

Dacă ε este un enunț, iar ψ este o FNC prenex cu $\varepsilon \models\!\!\!\dashv \psi$, atunci mulțimea de clauze care îl compun pe ψ se numește *formă clauzală* pentru ε.
Dacă $n \in \mathbb{N}^*$, iar $\varepsilon_1, \ldots, \varepsilon_n$ sunt enunțuri, atunci reuniunea unor forme clauzale pentru $\varepsilon_1, \ldots, \varepsilon_n$ se numește *formă clauzală* pentru $\{\varepsilon_1, \ldots, \varepsilon_n\}$.

Definiție

- **Clauza vidă** (i. e. clauza fără literali, clauza fără elemente) se notează cu $\square$ (pentru a o deosebi de **mulțimea vidă de clauze**, $\emptyset$).
- O clauză C se zice *trivială* ddacă există o formulă atomică α astfel încât $\alpha, \neg\alpha \in C$.
- O clauză nevidă $C = \{L_1, \ldots, L_n\}$ (cu $n \in \mathbb{N}^*$ și $L_1, \ldots, L_n$ literali) se zice *satisfiabilă* ddacă enunțul $L_1 \vee \ldots \vee L_n$ corespunzător lui C e satisfiabil.
- O mulțime finită de clauze se zice *satisfiabilă* ddacă enunțul în FNC corespunzător acelei mulțimi de clauze e satisfiabil.

Caracterizarea deducțiilor semantice pe care se bazează rezolvarea interogărilor de către Prolog prin tehnica rezoluției

Remarcă

Fie $n \in \mathbb{N}^*$, iar $\varphi_1, \ldots, \varphi_n$ și φ enunțuri.
Să observăm că are loc echivalența:

$$\models \varphi \text{ ddacă } \neg\varphi \text{ e nesatisfiabil (adică nu are model).}$$

Într–adevăr: $\models \varphi$ (i.e. φ e adevăr semantic) ddacă, pentru orice algebră $\mathcal{A}$ de signatură τ, avem $||\varphi||_{\mathcal{A}} = 1$ ddacă, pentru orice algebră $\mathcal{A}$ de signatură τ, avem $||\neg\varphi||_{\mathcal{A}} = \overline{||\varphi||_{\mathcal{A}}} = \overline{1} = 0$ ddacă $\neg\varphi$ e nesatisfiabil.
Mai mult, ca o consecință a Teoremei Deducției și a observației de mai sus, avem următoarea caracterizare a deducției semantice $\{\varphi_1, \ldots, \varphi_n\} \models \varphi$:

$$\{\varphi_1, \ldots, \varphi_n\} \models \varphi \text{ ddacă } \{\varphi_1 \wedge \ldots \wedge \varphi_n\} \models \varphi$$

$$\text{ddacă } \models (\varphi_1 \wedge \ldots \wedge \varphi_n) \rightarrow \varphi$$

$$\text{ddacă enunțul } \neg[(\varphi_1 \wedge \ldots \wedge \varphi_n) \rightarrow \varphi] \text{ e nesatisfiabil.}$$

Definiție (definim substituțiile pe clauze – la fel putem proceda pentru FNC fără cuantificatori reprezentate ca mulțimi de clauze)

Fie $\sigma : Term(\mathcal{L}_\tau) \to Term(\mathcal{L}_\tau)$ o substituție.
Pentru orice clauză $C = \{L_1, \ldots, L_p\}$, unde $p \in \mathbb{N}^*$, iar $L_1, \ldots, L_p$ sunt literali, definim $\sigma(C) = \{\sigma(L_1), \ldots, \sigma(L_p)\}$.

Remarcă (la fel va fi în cazul FNC fără cuantificatori)

Cu notațiile din definiția anterioară, clauza $\sigma(C) = \{\sigma(L_1), \ldots, \sigma(L_p)\}$ corespunde enunțului $\sigma(L_1) \vee \ldots \vee \sigma(L_p) = \sigma(L_1 \vee \ldots \vee L_p)$ dat de valoarea lui σ în enunțul $L_1 \vee \ldots \vee L_p$ corespunzător clauzei C.

Notație

Fie $n \in \mathbb{N}^*$, $x_1, \ldots, x_n \in Var$, două câte două distincte, și $t_1, \ldots, t_n \in Term(\mathcal{L}_\tau)$.
Notăm cu

$$\{x_1/t_1, \ldots, x_n/t_n\} : Var \to Term(\mathcal{L}_\tau)$$

substituția definită prin:

$$\begin{cases} (\forall\, i \in \overline{1, n})\, (\{x_1/t_1, \ldots, x_n/t_n\}(x_i) = t_i); \\ (\forall\, x \in Var \setminus \{x_1, \ldots, x_n\})\, (\{x_1/t_1, \ldots, x_n/t_n\}(x) = x). \end{cases}$$

Notație

Fie $n, k \in \mathbb{N}^*$, $x_1, \ldots, x_n \in Var$, două câte două distincte, $i_1, \ldots, i_k \in \overline{1, n}$, astfel încât $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, $t_{i_1}, \ldots, t_{i_k} \in Term(\mathcal{L}_\tau)$ și $\varphi(x_1, \ldots, x_n)$ o formulă fără cuantificatori cu $V(\varphi) \subseteq \{x_1, \ldots, x_n\}$.
Atunci notăm cu

$$\varphi(x_1, \ldots, x_{i_1-1}, t_{i_1}, x_{i_1+1}, \ldots, x_{i_2-1}, t_{i_2}, x_{i_2+1}, \ldots, x_{i_k-1}, t_{i_k}, x_{i_k+1}, \ldots, x_n)$$

formula $\{x_{i_1}/t_{i_1}, \ldots, x_{i_k}/t_{i_k}\}(\varphi)$.

Exemplu

Dacă v, x, y, z sunt variabile distincte, s, t, u sunt termeni, ρ este un simbol de relație binară, f e un simbol de operație binară, g e un simbol de operație unară și c este un simbol de constantă, atunci rezultatul aplicării substituției $\{x/s, y/t, z/u\}$ asupra formulei fără cuantificatori:

$$\varphi(v, x, y, z) = \neg\,[\rho(g(x), f(v, y)) \to \neg\,\rho(f(g(y), g(z)), f(v, g(c))]$$

este formula fără cuantificatori: $\{x/s, y/t, z/u\}(\varphi(v, x, y, z)) = \varphi(v, s, t, u) =$

$$\neg\,[\rho(g(s), f(v, t)) \to \neg\,\rho(f(g(t), g(u)), f(v, g(c))].$$

Pentru a aplica rezoluția unui enunț, enunțul trebuie pus într–un anumit tip de FNC, numit **formă Skolem**, despre care se poate demonstra că există pentru orice enunț și satisfiabilă ddacă acel enunț e satisfiabil.
Fie ε un enunț. Folosind proprietățile de mai sus și, eventual, redenumind variabilele pentru a nu avea variabile cuantificate și universal, și existențial (mai multe detalii în SEMINAR), se determină o formă prenex pentru ε:

$$\varepsilon \models \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n \varphi(x_1, \ldots, x_n),$$

unde $n \in \mathbb{N}^*$, $x_1, \ldots, x_n \in Var$, $\mathcal{Q}_1, \ldots, \mathcal{Q}_n \in \{\forall, \exists\}$ și $\varphi(x_1, \ldots, x_n)$ este o formulă fără cuantificatori. Apoi, folosind proprietățile 1–9 de mai sus, folosite în calculul propozițional pentru a pune enunțurile în FNC, se pune $\varphi(x_1, \ldots, x_n)$ într–o FNC ψ, astfel obținându–se o altă formă prenex pentru ε:

$$\varepsilon \models \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n \psi(x_1, \ldots, x_n),$$

cu formula fără cuantificatori ψ în FNC.
Pentru fiecare cuantificator existențial $\mathcal{Q}_i = \exists$, cu $i \in \overline{1, n}$, se adaugă la signatura τ câte o operație "fictivă" h_i, numită *funcție Skolem*, de aritate $i - |\{j \in \overline{1, i} \mid \mathcal{Q}_j = \exists\}|$ (astfel că, în cazul în care $\mathcal{Q}_1 = \exists$, h_1 este o constantă), și se înlocuiește fiecare apariție a variabilei x_i în $\psi(x_1, \ldots, x_n)$ cu termenul h_i având $V(h_i) = \{x_1, \ldots, x_i\} \setminus \{x_j \mid j \in \overline{1, i}\}$, iar cuantificatorii existențiali sunt eliminați din forma prenex anterioară.

Astfel obținem următoarea FNC: $\chi = \forall x_1, \ldots \forall x_{i_1-1} \forall x_{i_1+1} \ldots \forall x_{i_2-1} \forall x_{i_2+1} \ldots \forall x_{i_k-1} \forall x_{i_k+1} \ldots \forall x_n \, \varphi(x_1, \ldots, x_{i_1-1}, h_{i_1}(x_1, \ldots, x_{i_1-1}), x_{i_1+1}, \ldots, x_{i_2-1}, h_{i_2}(x_1, \ldots, x_{i_1-1}, x_{i_1+1}, \ldots, x_{i_2-1}), x_{i_2+1}, \ldots, x_{i_k-1}, h_{i_k}(x_1, \ldots, x_{i_1-1}, x_{i_1+1}, \ldots, x_{i_2-1}, x_{i_2+1}, \ldots, x_{i_k-1}), x_{i_k+1}, \ldots, x_n)$.
Fie $p = n - k$ și (doar pentru a simplifica următoarea scriere) să redenumim variabilele $x_1, \ldots, x_{i_1-1}, x_{i_1+1}, \ldots, x_{i_2-1}, x_{i_2+1}, \ldots, x_{i_k-1}, x_{i_k+1}, \ldots, x_n$ în $y_1, \ldots, y_p$, respectiv. Dacă $C_1(y_1, \ldots, y_p), \ldots, C_r(y_1, \ldots, y_p)$ sunt clauzele formulei fără cuantificatori în FNC $\varphi(x_1, \ldots, x_{i_1-1}, h_{i_1}(x_1, \ldots, x_{i_1-1}), x_{i_1+1}, \ldots, x_{i_2-1}, h_{i_2}(x_1, \ldots, x_{i_1-1}, x_{i_1+1}, \ldots, x_{i_2-1}), x_{i_2+1}, \ldots, x_{i_k-1}, h_{i_k}(x_1, \ldots, x_{i_1-1}, x_{i_1+1}, \ldots, x_{i_2-1}, x_{i_2+1}, \ldots, x_{i_k-1}), x_{i_k+1}, \ldots, x_n)$, atunci:

$$\chi = \forall y_1 \ldots \forall y_p \, C_1(y_1, \ldots, y_p) \wedge \ldots \wedge C_r(y_1, \ldots, y_p) \;\models\!\!\dashv$$

$$(\forall y_1 \ldots \forall y_p \, C_1(y_1, \ldots, y_p)) \wedge \ldots \wedge (\forall y_1 \ldots \forall y_p \, C_r(y_1, \ldots, y_p)),$$

iar această din urmă formulă este satisfiabilă ddacă fiecare dintre formulele $\forall y_1 \ldots \forall y_p \, C_1(y_1, \ldots, y_p)$, ..., $\forall y_1 \ldots \forall y_p \, C_r(y_1, \ldots, y_p)$ este satisfiabilă.
Acum redenumim variabilele $\{y_1, \ldots, y_p\}$ în fiecare dintre clauzele $C_1, \ldots, C_r$ astfel încât mulțimile $V(C_1), \ldots, V(C_r)$ să devină două câte două disjuncte: $V(C_1) = \{z_{11}, \ldots, z_{1p}\}$, ..., $V(C_r) = \{z_{r1}, \ldots, z_{rp}\}$, și, folosind, ca și mai sus, distributivitatea cuantificatorului universal ($\forall$) față de conjuncție ($\wedge$), obținem:

$$\xi \models\!\!\dashv \gamma = \forall z_{11} \ldots \forall z_{1p} \ldots \forall z_{r1} \ldots \forall z_{rp}(C_1(z_{11}, \ldots, z_{1p}) \wedge \ldots \wedge C_r(z_{r1}, \ldots, z_{rp})).$$

FNC γ se numește *formă Skolem* pentru ε, și este satisfiabilă ddacă ε e satisfiabil. Nu putem spune că $\varepsilon \models\!\!\!\dashv \gamma$, întrucât ε și γ nu sunt satisfăcute de aceleași algebre: la signatura τ trebuie să adăugăm simboluri de operații/constante corespunzătoare funcțiilor Skolem pentru a obține signatura algebrelor în care este evaluată valoarea de adevăr a enunțului γ.

Definiție (**unificare** între **literali**)

Prin *unificare* între **formule atomice** înțelegem unificarea acelor formule privite ca termeni cu operația dominantă de rezultat boolean asociată relației din acele formule atomice, ca în Prolog.
Dacă φ și ψ sunt formule atomice, vom spune că $\neg\varphi$ și $\neg\psi$ *unifică* ddacă φ și ψ unifică. Un *(cel mai general) unificator* pentru $\neg\varphi$ și $\neg\psi$ este un *(cel mai general) unificator* pentru φ și ψ, adică un (cel mai general) unificator pentru termenii obținuți din φ și ψ prin înlocuirea relațiilor din aceste formule atomice cu operațiile de rezultat boolean asociate acestor relații.

Definiție (**clauze triviale**)

Numim *clauză trivială* o clauză care conține doi literali φ și $\neg\psi$, unde φ și ψ sunt formule atomice care **unifică**.

(Rezoluția (regulă de deducție pentru logica clasică a predicatelor))

Pentru orice clauze C și D cu $V(C) \cap V(D) = \emptyset$, dacă φ și ψ sunt formule atomice astfel încât $\varphi, \neg\varphi$ nu unifică, cu niciun literal din C și $\psi, \neg\psi$ nu unifică, cu niciun literal din D, iar $\sigma : Term(\mathcal{L}_\tau) \to Term(\mathcal{L}_\tau)$ este o substituție cu proprietatea că $\sigma(\varphi) = \sigma(\psi)$, atunci:

$$\frac{C \cup \{\varphi\}, D \cup \{\neg\psi\}}{\sigma(C) \cup \sigma(D)}.$$

Definiție (derivări prin rezoluție)

Fie o mulțime finită de clauze $\{D_1, \ldots, D_k\}$.
Dacă $i, j \in \overline{1, k}$ a. î. $i \neq j$ și există două formule atomice φ și ψ astfel încât $\varphi, \neg\varphi$ nu unifică, cu niciun literal din D_i, $\psi, \neg\psi$ nu unifică, cu niciun literal din D_j, iar $\sigma : Term(\mathcal{L}_\tau) \to Term(\mathcal{L}_\tau)$ este o substituție cu proprietatea că $\sigma(\varphi) = \sigma(\psi)$, atunci mulțimea de clauze $R = \{\sigma(D_1), \ldots, \sigma(D_k)\}$ se numește *rezolvent* al mulțimii de clauze $Q = \{D_i \cup \{\varphi\}, D_j \cup \{\neg\psi\}\} \cup \{D_t \mid t \in \overline{1, k} \setminus \{i, j\}\}$.
Deducția $\frac{Q}{R}$ se numește *derivare prin rezoluție* a mulțimii Q. Vom numi orice succesiune de derivări prin rezoluție tot *derivare prin rezoluție*. O succesiune de derivări prin rezoluție care începe cu o FNC/mulțime de clauze μ și se termină cu o FNC/mulțime de clauze ν se numește *derivare prin rezoluție a lui ν din μ*.

(Algoritmul Davis–Putnam (abreviat *DP*) pentru Logica Predicatelor)

INPUT: $m \in \mathbb{N}^*$ și o mulțime $S = \{C_1, \ldots, C_m\}$ de clauze netriviale având $V(C) \cap V(D) = \emptyset$ pentru orice clauze $C, D \in S$ cu $C \neq D$;

$i := 1$; $S_1 := S$;

PASUL 1: considerăm o formulă atomică α_i care apare în măcar una dintre clauzele din S_i;

$T_i^0 := \{j \in \overline{1, m} \mid C_j \in S_i, (\exists\, \beta_i)(\neg\, \beta_{i,j} \in C_j$ și $\beta_{i,j}$ unifică, cu $\alpha_i)\}$;

$T_i^1 := \{j \in \overline{1, m} \mid C_j \in S_i, (\exists\, \beta_i)(\beta_{i,j} \in C_j$ și $\beta_{i,j}$ unifică, cu $\alpha_i)\}$;

$T_i := T_i^0 \cup T_i^1$;

fie σ_i un unificator pentru elementele mulțimii $\{\beta_{i,j} \mid j \in T_i\}$;

PASUL 2: dacă $T_i^0 \neq \emptyset$ și $T_i^1 \neq \emptyset$,

atunci $U_i := \{\sigma_i(C_j \setminus \{\neg\, \beta_{i,j}\}) \cup \sigma_i(C_k \setminus \{\beta_{i,k}\}) \mid j \in T_i^0, k \in T_i^1\}$;

altfel $U_i := \emptyset$;

PASUL 3: $S_{i+1} := (S_i \setminus \{C_j \mid j \in T_i\}) \cup U_i$;

$S_{i+1} := S_{i+1} \setminus \{C \in S_{i+1} \mid (\exists\, \beta \in V)\,(\beta$ formulă atomică și $\beta, \neg\, \beta \in C)\}$ (eliminăm din S_{i+1} clauzele triviale);

PASUL 4: dacă $S_{i+1} = \emptyset$,

atunci OUTPUT: S *e satisfiabilă*;

altfel, dacă $\square \in S_{i+1}$,

atunci OUTPUT: S *nu e satisfiabilă*;

altfel $i := i + 1$ și mergi la PASUL 1.

În derivările prin rezoluție, inclusiv în unificările efectuate în cadrul acestor derivări, deci și în aplicarea algoritmului DP, funcțiile Skolem sunt tratate ca orice funcții. Rezultatul algoritmului DP e influențat de FORMA SKOLEM obținută pentru enunțul despre care dorim să aflăm dacă e satisfiabil sau nu. A se vedea un exemplu mai jos.

Pentru **tipurile de enunțuri** pentru care **Prologul** aplică tehnica rezoluției, algoritmul DP este **corect și complet**: se termină într–un număr finit de pași și dă rezultatul corect: determină dacă un astfel de enunț e satisfiabil sau nu.
În general însă, spre deosebire de cazul logicii propozițiilor, rezultatul aplicării algoritmului DP unui enunț cu sau fără cuantificatori existențiali nu este neapărat corect.

Revenind la o **formă Skolem** pentru un enunț ε: ce rol au funcțiile Skolem?

Dacă $x, y \in Var$, iar φ este o formulă cu $FV(\varphi) \subseteq \{x, y\}$, de ce avem: $\exists x \forall y \varphi \models \forall y \exists x \varphi$, dar nu și $\forall y \exists x \varphi \not\models \exists x \forall y \varphi$? Pentru că, în enunțul $\forall y \exists x \varphi$, o valoare a a lui x care satisface acest enunț poate să depindă de valoarea b a lui y pentru care perechea (a, b) satisface acest enunț.

Exemplu

- $(\exists x \in \mathbb{R})(\forall y \in \mathbb{R})(x + y = 0)$: **fals**: nu există un astfel de x comun tuturor valorilor lui $y \in \mathbb{R}$;
- $(\forall y \in \mathbb{R})(\exists x \in \mathbb{R})(x + y = 0)$: **adevărat**;

Exemplu (continuare)

- $(\exists\, x \in \mathbb{N})\,(\forall\, y \in \mathbb{Z})\,(x \in \{y, -y\})$: **fals**: nu există un astfel de x comun tuturor valorilor lui $y \in \mathbb{Z}$;
- $(\forall\, y \in \mathbb{Z})\,(\exists\, x \in \mathbb{N})\,(x \in \{y, -y\})$: **adevărat**.

Fie $f : \mathbb{R} \to \mathbb{R}$, având cel puțin 3 valori distincte, i.e. cardinalul imaginii sale $f(\mathbb{R})$ mai mare sau egal cu 3.

- $(\exists\, x \in [0, +\infty))\,(\forall\, y \in \mathbb{R})\,(x \in \{f(y), -f(y)\})$: **fals**: nu există un astfel de x comun tuturor valorilor lui $y \in \mathbb{R}$, pentru că alegerea lui f ne asigură de faptul că există $y_0, y_1 \in \mathbb{R}$ cu proprietatea că $f(y_1) \notin \{f(y_0), -f(y_0)\}$, așadar perechea $(x_1 = f(y_1), y_0)$ nu satisface proprietatea $x_1 \in \{f(y_0), -f(y_0)\}$;
- $(\forall\, y \in \mathbb{R})\,(\exists\, x \in [0, +\infty))\,(x \in \{f(y), -f(y)\})$: **adevărat**.

O **formă Skolem** pentru enunțul $(\forall\, y)\,(\exists\, x)\,(x + y = 0)$ este:
$(\forall\, y)\,(g(y) + y = 0)$, unde g este o funcție Skolem unară, iar acest enunț în formă Skolem, pentru domeniul valorilor $\mathbb{R}$ pentru ambele variabile, x și y, semnifică faptul că există o funcție $g : \mathbb{R} \to \mathbb{R}$ astfel încât $(\forall\, y \in \mathbb{R})\,(g(y) + y = 0)$.
O **formă Skolem** pentru enunțul $(\forall\, y)\,(\exists\, x)\,(x \in \{y, -y\})$ este:
$(\forall\, y)\,(h(y) \in \{y, -y\})$, unde h este o funcție Skolem unară, iar acest enunț în formă Skolem, pentru domeniile valorilor $\mathbb{N}$, respectiv $\mathbb{Z}$ pentru variabilele x, respectiv y, semnifică faptul că există o funcție $h : \mathbb{Z} \to \mathbb{N}$ astfel încât $(\forall\, y \in \mathbb{Z})\,(h(y) \in \{y, -y\})$.

O **formă Skolem** pentru enunțul $(\forall\, y)\,(\exists\, x)\,(x \in \{f(y), -f(y)\})$ este: $(\forall\, y)\,(k(y) \in \{f(y), -f(y)\})$, unde k este o funcție Skolem unară, iar acest enunț în formă Skolem, pentru pentru domeniile valorilor $[0, +\infty)$, respectiv $\mathbb{R}$ pentru variabilele x, respectiv y, semnifică faptul că există o funcție $k : \mathbb{R} \to [0, +\infty)$ astfel încât $(\forall\, y \in \mathbb{R})\,(k(y) \in \{f(y), -f(y)\})$. Într–adevăr, acest enunț este satisfăcut de funcția $k : \mathbb{R} \to [0, +\infty)$, definită prin:

$$(\forall\, y \in \mathbb{R})\left(k(y) = |f(y)| = \begin{cases} f(y), & \text{dacă } f(y) \geq 0, \\ -f(y), & \text{altfel.} \end{cases}\right).$$

Deci faptul că funcția k nu coincide nici cu f, nici cu $-$ nu influențează satisfiabilitatea enunțului anterior.

Exemplu

Să considerăm un limbaj de ordinul I conținând două simboluri de relații unare p și q. Fie x, y variabile distincte. Considerăm enunțurile:

- $\varphi = \forall\, x\, p(x) \wedge \exists\, y\, q(y)$;
- $\psi = \exists\, x\, [(p(x) \vee q(x)) \wedge \forall\, y\, ((p(x) \vee \neg\, q(y)) \wedge \neg\, p(x))]$.

$\varphi \models\!\!\!

Așadar două forme Skolem diferite pentru enunțul φ sunt:

- $\forall x\,(p(x) \wedge q(c))$, unde c este o constantă Skolem; această formă Skolem corespunde mulțimii de clauze $\{\{p(x)\}, \{q(c)\}\}$, care nu are derivări prin rezoluție, deci algoritmul DP determină satisfiabilitatea acestei mulțimi de clauze;
- $\forall x\,(p(x) \wedge q(f(x)))$, unde f este o funcție Skolem unară; această formă Skolem corespunde mulțimii de clauze $\{\{p(x)\}, \{q(f(x))\}\}$, care nu are derivări prin rezoluție, deci algoritmul DP determină și satisfiabilitatea acestei mulțimi de clauze.

La fel ca mai sus:

- $\psi \models\!\!\!\!\!\!\;\;\; \exists x \forall y\,[(p(x) \vee q(x)) \wedge (p(x) \vee \neg q(y)) \wedge \neg p(x)]$, având drept formă Skolem enunțul $\forall y\,[(p(c) \vee q(c)) \wedge (p(c) \vee \neg q(y)) \wedge \neg p(c)]$, unde c este o constantă Skolem, corespunzător mulțimii de clauze $\{\{p(c), q(c)\}, \{p(c), \neg q(y)\}, \{\neg p(c)\}\}$;
- $\psi \models\!\!\!\!\!\!\;\;\; \forall y \exists x\,[(p(x) \vee q(x)) \wedge (p(x) \vee \neg q(y)) \wedge \neg p(x)]$, având drept formă Skolem enunțul $\forall y\,[(p(f(y)) \vee q(f(y))) \wedge (p(f(y)) \vee \neg q(y)) \wedge \neg p(f(y))]$, unde f este o funcție Skolem unară, corespunzător mulțimii de clauze $\{\{p(f(v)), q(f(v))\}, \{p(f(y)), \neg q(y)\}, \{\neg p(f(z))\}\}$.

Pentru prima dintre mulțimile de clauze de mai sus avem următoarea derivare prin rezoluție a clauzei vide, prin urmare algoritmul DP determină nesatisfiabilitatea acestei mulțimi de clauze:

$$\dfrac{\dfrac{\{p(c), q(\cancel{c})\}, \{p(c), \neg q(\cancel{y})\} \text{ (cu unificatorul } \{y/c\}), \{\neg p(c)\}}{\{p(\cancel{c})\}, \{\neg p(\cancel{c})\}}}{\square}$$

Pentru a doua dintre mulțimile de clauze de mai sus avem următoarele derivări prin rezoluție, dintre care niciuna nu ajunge la clauza vidă, prin urmare algoritmul DP determină satisfiabilitatea acestei mulțimi de clauze:

$$\dfrac{\dfrac{\{p(f(v)), q(f(\cancel{v}))\}, \{p(f(y)), \neg q(\cancel{y})\} \text{ (cu unificatorul } \{y/f(v)\}), \{\neg p(f(z))\}}{\{p(f(\cancel{v})), p(f(f(v)))\}, \{\neg p(f(\cancel{z}))\} \text{ (cu unificatorul } \{z/v\})}}{\{p(f(f(v)))\}}$$

$$\dfrac{\dfrac{\{p(f(v)), q(f(\cancel{v}))\}, \{p(f(y)), \neg q(\cancel{y})\} \text{ (cu unificatorul } \{y/f(v)\}), \{\neg p(f(z))\}}{\{p(f(v)), p(f(f(\cancel{v})))\}, \{\neg p(f(\cancel{z}))\} \text{ (cu unificatorul } \{z/f(v)\})}}{\{p(f(v))\}}$$

$$\dfrac{\dfrac{\{p(f(\cancel{v})), q(f(v))\}, \{p(f(y)), \neg q(y)\}, \{\neg p(f(\cancel{z}))\} \text{ (cu unificatorul } \{z/v\})}{\{q(f(\cancel{v}))\}, \{p(f(y)), \neg q(\cancel{y})\} \text{ (cu unificatorul } \{y/f(v)\})}}{\{p(f(f(v)))\}}$$

Vom vedea că Prologul aplică rezoluția numai pentru FNC prenex (FNC fără cuantificatori precedate numai de cuantificatori universali) obținute fără Skolemizare (i.e. fără înlocuirea unor variabile cu funcții Skolem), mai precis pentru un tip particular de astfel de FNC prenex: **Prologul** întotdeauna aplică **regula rezoluției** pentru o **clauză Horn** și o **clauză scop**: definițiile mai jos.

1 Introducere

2 Inițiere în programarea în Prolog

3 Să recapitulăm noțiunile din Logica Clasică a Predicatelor care stau la baza funcționării Prologului

4 Să exemplificăm noțiunile anterioare într–un program în Prolog

5 Algoritmul de unificare

6 Cu ce fel de signaturi și structuri algebrice de acele signaturi și cu ce tipuri/cazuri particulare de formule lucrează limbajul Prolog?

7 Mnemonic din sintaxa calculului cu predicate clasic

8 Modele pentru formule, satisfiabilitate în Logica Clasică a Predicatelor

9 Rezoluția în Logica Clasică a Predicatelor – regula de deducție care stă la baza funcționării limbajului Prolog

10 Rezoluția în Prolog

Amintesc că:

- *formulele atomice* sunt formulele fără cuantificatori și fără conectori logici;
- *literalii* sunt formulele atomice și negațiile formulelor atomice;
- *clauzele* sunt disjuncțiile finite (nu neapărat nevide) de literali; clauzele se identifică, cu mulțimile literalilor pe care îi conțin; dacă sunt nevide, clauzele sunt formule fără cuantificatori;
- *FNC prenex* sunt enunțurile date de conjuncții finite și nevide de clauze nevide precedate de cuantificatori universali; FNC prenex se identifică, cu mulțimile clauzelor pe care le conțin.

Definiție

O *clauză Horn* sau *clauză definită* este o clauză care conține exact o formulă atomică nenegată (implicit e o clauză nevidă), iar ceilalți literali pe care îi conține sunt formule atomice negate.
O *clauză scop* este o clauză nevidă care conține numai formule atomice negate.

Amintesc cu ce tipuri de formule (mai precis enunțuri) lucrează Prologul:

- **faptele** corespund unor enunțuri de tipul următor: $\forall x_1 \forall x_2 \ldots \forall x_n \alpha$, unde $n \in \mathbb{N}^*$ și α este o formulă atomică;
- **regulile** corespund unor enunțuri de tipul:$\forall y_1 \forall y_2 \ldots \forall y_k (\varphi \to \beta)$, unde $k \in \mathbb{N}^*$, φ este o formulă fără cuantificatori, iar β este o formulă atomică;
- **interogările** corespund unor formule de tipul $\exists z_1 \exists z_2 \ldots \exists z_p \psi$, unde $p \in \mathbb{N}^*$ și ψ este o formulă fără cuantificatori.

Pentru cele ce urmează, cu notațiile de mai sus, vom presupune că φ și ψ sunt conjuncții finite (desigur, nevide) de formule atomice.
Dacă în φ sau ψ apare negația unei formule atomice, de exemplu $\rho = \neg R(t_1, \ldots, t_m)$, unde $m \in \mathbb{N}^*$, $t_1, \ldots, t_m$ sunt termeni și R este un simbol de relație m–ară, atunci transformăm pe ρ într–o formulă atomică introducând în signatură simbolul de relație m–ară $\neg R$ astfel încât, în orice algebră $\mathcal{A}$ de signatura definită de respectivul program în Prolog care satisface enunțurile date de faptele și regulile din programul respectiv, dacă A este mulțimea de elemente ale lui $\mathcal{A}$, atunci $(\neg R)^{\mathcal{A}} = A^m \setminus R^{\mathcal{A}} = \{(a_1, \ldots, a_m) \in A^m \mid (a_1, \ldots, a_m) \notin R\}$.
Se adaptează teoria următoare și pentru cazul în care, în formule precum φ, ψ apar disjuncții de formule atomice, nu numai conjuncții. De exemplu, o regulă de forma:

concluzie :- *posibilitatea*1; *posibilitatea*2.

se poate înlocui cu următoarele două reguli:

concluzie :- *posibilitatea*1.
concluzie :- *posibilitatea*1.

O regulă de forma următoare, de exemplu:

concluzia :- *premisa*1, (*posibilitate*1; *posibilitate*2; *posibilitate*3), *premisa*2.

se poate înlocui cu următoarele trei reguli:

concluzia :- *premisa*1, *posibilitate*1, *premisa*2.
concluzia :- *premisa*1, *posibilitate*2, *premisa*2.
concluzia :- *premisa*1, *posibilitate*3, *premisa*2.

Un scop de forma următoare, de exemplu:

?- *subscop*1, (*varianta*1; *varianta*2), *subscop*2, *subscop*3.

e satisfăcut ddacă e satisfăcut măcar unul dintre scopurile:

?- *subscop*1, *varianta*1, *subscop*2, *subscop*3.
?- *subscop*1, *varianta*2, *subscop*2, *subscop*3.

iar soluțiile primei interogări de mai sus sunt date de reuniunea soluțiilor celor două interogări anterioare.
Să presupunem, așadar, că: $\varphi = \gamma_1 \wedge \ldots \wedge \gamma_q$ și $\psi = \delta_1 \wedge \ldots \wedge \delta_s$, unde $q, s \in \mathbb{N}^*$, iar $\gamma_1, \ldots, \gamma_q, \delta_1, \ldots, \delta_s$ sunt formule atomice. Atunci:

$$\forall\, y_1 \forall\, y_2 \ldots \forall\, y_k\, (\varphi \to \beta) = \forall\, y_1 \forall\, y_2 \ldots \forall\, y_k\, ((\gamma_1 \wedge \ldots \wedge \gamma_q) \to \beta) \models\!\!\!\dashv$$

$$\forall\, y_1 \forall\, y_2 \ldots \forall\, y_k\, (\neg\, (\gamma_1 \wedge \ldots \wedge \gamma_q) \vee \beta) \models\!\!\!\dashv \forall\, y_1 \forall\, y_2 \ldots \forall\, y_k\, (\neg\, \gamma_1 \vee \ldots \vee \neg\, \gamma_q \vee \beta),$$

iar $\neg\, \gamma_1 \vee \ldots \vee \neg\, \gamma_q \vee \beta$ este o **clauză Horn**.
Cum α este o formulă atomică, în particular α este o **clauză Horn**.

Așadar **clauzele** din **baza de cunoștințe** constau în enunțuri date de **clauze Horn** precedate de cuantificatori universali.

Amintesc că, dacă baza de cunoștințe este formată din enunțurile $\varepsilon_1, \ldots, \varepsilon_r$, cu $r \in \mathbb{N}^*$, iar ε este un enunț corespunzător unei interogări, atunci cerința adresată Prologului prin această interogare este de a determina dacă:

$$\{\varepsilon_1, \ldots, \varepsilon_r\} \models \varepsilon,$$

sau, echivalent:

$$\{\varepsilon_1 \wedge \ldots \wedge \varepsilon_r\} \models \varepsilon,$$

sau, echivalent, conform Teoremei Deducției Semantice:

$$\models (\varepsilon_1 \wedge \ldots \wedge \varepsilon_r) \rightarrow \varepsilon,$$

fapt echivalent cu:

enunțul $\neg\,[(\varepsilon_1 \wedge \ldots \wedge \varepsilon_r) \rightarrow \varepsilon]$ e nesatisfiabil,
adică enunțul $\varepsilon_1 \wedge \ldots \wedge \varepsilon_r \wedge \neg\,\varepsilon$ e nesatisfiabil,

întrucât

$$\neg\,[(\varepsilon_1 \wedge \ldots \wedge \varepsilon_r) \rightarrow \varepsilon] \models\!\!\!\dashv \neg\,[\neg\,(\varepsilon_1 \wedge \ldots \wedge \varepsilon_r) \vee \varepsilon] \models\!\!\!\dashv \varepsilon_1 \wedge \ldots \wedge \varepsilon_r \wedge \neg\,\varepsilon,$$

iar acest din urmă enunț poate fi transformat într–o **FNC prenex** prin mutarea cuantificatorilor în față (a se vedea mai jos).

Definiție

O derivare prin rezoluție a clauzei vide $\square$ din FNC prenex $\varepsilon_1 \wedge \ldots \wedge \varepsilon_r \wedge \neg \varepsilon$ se numește *SLD–respingere* a clauzei ε din mulțimea de clauze $\{\varepsilon_1, \ldots, \varepsilon_r\}$.

(Principala regulă de deducție utilizată de Prolog)

Pentru a satisface scopul ε, Prologul caută o SLD–respingere a clauzei ε din mulțimea de clauze corespunzătoare bazei de cunoștințe.
Acest caz al rezoluției aplicat de Prolog este numit *rezoluție SLD* (*Selective Linear Definite Clause Resolution*: rezoluție liniară selectivă pe clauze definite).

Baza de cunoștințe se identifică, cu, conjuncția enunțurilor corespunzătoare faptelor și regulilor ($\varepsilon_1 \wedge \ldots \wedge \varepsilon_r$ cu notația de mai sus), care, prin mutarea cuantificatorilor în față, devine o **FNC prenex** constând din **clauze Horn**.

Într–adevăr, de exemplu, considerând faptul și regula de mai sus, dacă mulțimea de variabile $\{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_k\} = \{v_1, \ldots, v_j\}$, avem:

$$\forall x_1 \ldots \forall x_n \, \alpha \wedge \forall y_1 \ldots \forall y_k \, (\varphi \to \beta) \models\!\!\!\dashv \forall v_1 \ldots \forall v_j \, [\alpha \wedge (\varphi \to \beta)] =$$

$$\forall v_1 \ldots \forall v_j \, [\alpha \wedge ((\gamma_1 \wedge \ldots \wedge \gamma_q) \to \beta)] \models\!\!\!\dashv \forall v_1 \ldots \forall v_j \, [\alpha \wedge (\neg \gamma_1 \vee \ldots \vee \neg \gamma_q \vee \beta)].$$

La fel pentru mai multe fapte și/sau reguli.
De exemplu, dacă baza de cunoștințe este formată din faptul și regula de mai sus, iar $\{v_1, \ldots, v_j\} \cup \{z_1, \ldots, z_p\} = \{w_1, \ldots, w_l\}$ atunci interogarea de mai sus presupune a verifica nesatisfiabilitatea enunțului:

$$\forall x_1 \ldots \forall x_n \alpha \wedge \forall y_1 \ldots \forall y_k (\varphi \to \beta) \wedge \neg (\exists z_1 \exists z_2 \ldots \exists z_p \psi) \models\!\!\!\dashv$$

$$\forall x_1 \ldots \forall x_n \alpha \wedge \forall y_1 \ldots \forall y_k (\varphi \to \beta) \wedge \forall z_1 \ldots \forall z_p \neg \psi \models\!\!\!\dashv$$

$$\forall w_1 \ldots \forall w_l [\alpha \wedge (\varphi \to \beta) \wedge \neg \psi] \models\!\!\!\dashv \forall w_1 \ldots \forall w_l [\alpha \wedge (\varphi \to \beta) \wedge \neg (\delta_1 \wedge \ldots \wedge \delta_s)] \models\!\!\!\dashv$$

$$\forall w_1 \ldots \forall w_l (\alpha \wedge (\neg \gamma_1 \vee \ldots \vee \neg \gamma_q \vee \beta) \wedge (\neg \delta_1 \vee \ldots \vee \neg \delta_s)).$$

Observăm că negația interogării de mai sus, anume enunțul $\forall z_1 \ldots \forall z_p (\neg \delta_1 \vee \ldots \vee \neg \delta_s)$ este un enunț dat de o **clauză scop** precedată de cuantificatori universali.

Tehnica descrisă mai sus presupune adăugarea acestei clauze scop la mulțimea de clauze Horn corespunzătoare bazei de cunoștințe și aplicarea rezoluției pentru mulțimea de clauze astfel obținută:

- dacă această mulțime de clauze este **nesatisfiabilă**, atunci răspunsul la interogarea de mai sus este **true**: enunțul $\exists z_1 \ldots \exists z_p (\delta_1 \wedge \ldots \wedge \delta_s)$ se deduce semantic din baza de cunoștințe, deci scopul dat de acest enunț este satisfăcut;
- dacă această mulțime de clauze este **satisfiabilă**, atunci răspunsul la interogarea de mai sus este **false**: enunțul $\exists z_1 \ldots \exists z_p (\delta_1 \wedge \ldots \wedge \delta_s)$ nu este consecință semantică a bazei de cunoștințe, deci scopul dat de acest enunț eșuează.

Exemplu

Considerăm baza de cunoștințe:

$a.\ b.$

c :- $a, b.$

și interogarea: ?- c.

Regula din programul de mai sus corespunde formulei fără variabile:

$$(a \wedge b) \to c \models\!\!\!\dashv \neg(a \wedge b) \vee c \models\!\!\!\dashv \neg a \vee \neg b \vee c,$$

așadar întreaga bază de cunoștinte corespunde FNC:

$$a \wedge b \wedge [(a \wedge b) \to c] \models\!\!\!\dashv a \wedge b \wedge (\neg a \vee \neg b \vee c),$$

având forma clauzală: $\{\{a\}, \{b\}, \{\neg a, \neg b, c\}\}$.

Să vedem dacă, adăugând la această mulțime de clauze clauza $\{\neg c\}$, obținem o mulțime de clauze satisfiabilă:

$$\dfrac{\dfrac{\dfrac{\{a\}, \{b\}, \{\neg a, \neg b, \cancel{c}\}, \{\cancel{\neg c}\}}{\{a\}, \{\cancel{b}\}, \{\neg a, \cancel{\neg b}\}}}{\{\cancel{a}\}, \{\cancel{\neg a}\}}}{\square}$$

Am ajuns la **clauza vidă**, așadar mulțimea de clauze $\{\{a\}, \{b\}, \{\neg a, \neg b, c\}, \{\neg c\}\}$ e **nesatisfiabilă**, prin urmare:

$$\{a, b, (a \wedge b) \to c\} \models c,$$

Să ne amintim ce sunt **substituțiile**

adică scopul c e satisfăcut, deci răspunsul la interogarea ?- c este **true**.

Cum, în exemplul anterior, nu avem variabile, ne situăm în cazul **rezoluției propoziționale**. Cazul general al **rezoluției pentru logica predicatelor** presupune determinarea unui **unificator** la fiecare aplicare a **regulii rezoluției** și aplicarea acestui unificator la clauzele rămase după aplicarea acelui pas de rezoluție.

Definiție (o *substituție* este o funcție de la variabile la termeni)

O *substituție* este o funcție $\sigma : Var \to Term(\mathcal{L}_\tau)$.

Definiție și notație (prelungirea unei substituții la termeni)

Fie $\sigma : Var \to Term(\mathcal{L}_\tau)$. Următoarea extindere a lui σ la întreaga mulțime a termenilor se numește tot *substituție* (și, de obicei, se notează tot cu σ): $\widetilde{\sigma} : Term(\mathcal{L}_\tau) \to Term(\mathcal{L}_\tau)$, definită, recursiv, astfel:

- pentru orice $v \in Var$, $\widetilde{\sigma}(v) = \sigma(v)$;
- pentru orice (simbol de) constantă c, $\widetilde{\sigma}(c) = c$;
- pentru orice $h \in \mathbb{N}^*$, orice (simbol de) operație h–ară f și orice $t_1, \ldots, t_h \in Term(\mathcal{L}_\tau)$, $\widetilde{\sigma}(f(t_1, \ldots, t_h)) = f(\widetilde{\sigma}(t_1), \ldots, \widetilde{\sigma}(t_h))$.

Un **unificator** pentru o mulțime de termeni este o substituție care are aceeași valoare în acei termeni, i.e., aplicată acelor termeni, are ca rezultat termeni identici

Definiție

O *substituție* $\sigma : Var \to Term(\mathcal{L}_\tau)$ se numește *unificator* pentru $h \in \mathbb{N}^*$ termeni $t_1, \ldots, t_h \in Term(\mathcal{L}_\tau)$ ddacă $\widetilde{\sigma}(t_1) = \ldots = \widetilde{\sigma}(t_h)$.

Algoritmul de unificare aplicat unor termeni $t_1, \ldots, t_h$ ($h \in \mathbb{N}^*$) întoarce un **cel mai general unificator** al termenilor $t_1, \ldots, t_h$, adică un unificator μ al acestor termeni cu proprietatea că orice unificator al termenilor $t_1, \ldots, t_h$ este dat de compunerea unei substituții cu μ.

Amintesc și această:

Notație

Dacă $h \in \mathbb{N}^*$, $v_1, \ldots, v_h \in Var$ și $t_1, \ldots, t_h \in Term(\mathcal{L}_\tau)$, atunci substituția pe care o notăm $\{v_1/t_1, \ldots, v_h/t_h\} : Var \to Term(\mathcal{L}_\tau)$ este definită prin:

$$\begin{cases} \{v_1/t_1, \ldots, v_h/t_h\}(v_i) = t_i, & \text{pentru orice } i \in \overline{1, h}, \\ \{v_1/t_1, \ldots, v_h/t_h\}(v) = v, & \text{pentru orice } v \in Var \setminus \{v_1, \ldots, v_h\}. \end{cases}$$

Cu notațiile de mai sus, avem, în Prolog:

- **faptul** $\forall x_1 \forall x_2 \ldots \forall x_n \alpha$:

α. % poate fi privit ca o **regulă** cu **membrul stâng** α și **premisa vidă**

- **regula** $\forall y_1 \forall y_2 \ldots \forall y_k [(\gamma_1 \wedge \ldots \wedge \gamma_q) \rightarrow \beta]$:

β :- $\gamma_1, \ldots, \gamma_q$. % cu **membrul stâng** β și **premisa** $\gamma_1, \ldots, \gamma_q$

- **interogarea** $\exists z_1 \exists z_2 \ldots \exists z_p (\delta_1 \wedge \ldots \wedge \delta_s)$:

?- $\delta_1, \ldots, \delta_s$.

unde α, β, $\gamma_1, \ldots, \gamma_q$, $\delta_1, \ldots, \delta_s$ sunt **formule atomice**, adică relații aplicate la termeni, astfel că, în Prolog, pot fi considerate **termeni** cu operatorii dominanți dați de **operațiile de rezultat boolean asociate acelor relații** (a se vedea mai sus).

(Cum rezolvă Prologul interogarea ?- $\delta_1, \ldots, \delta_s$.?)

Prologul rezolvă o interogare ?- $\delta_1, \ldots, \delta_s$ (adică satisface **scopul compus** $\delta_1, \ldots, \delta_s$) satisfăcând, PE RÂND, **subscopurile** $\delta_1, \ldots, \delta_s$, DE LA STÂNGA LA DREAPTA.

Să detaliem modul în care Prologul răspunde interogărilor

Cum satisface Prologul un **subscop** δ_i, cu $i \in \overline{1, s}$? Considerând **faptele** și **regulile** din **baza de cunoștințe** DE SUS ÎN JOS și unificând (prin aplicarea **algoritmului de unificare**), cu un **cel mai general unificator** μ, pe δ_i cu **formula atomică** dintr–un:

- **fapt** α, caz în care următorul scop compus este:
  ?- $\mu(\delta_1), \ldots, \mu(\delta_{i-1}), \mu(\delta_{i+1}), \ldots, \mu(\delta_s)$.
  adică se scoate din scopul compus $\delta_1, \ldots, \delta_s$ subscopul δ_i, iar la celelalte subscopuri se aplică unificatorul μ; acesta este un **caz particular** al aplicării unei **reguli**, pentru **premisa vidă**;

sau din

- **membrul stâng** β **al unei reguli** β :- $\gamma_1, \ldots, \gamma_q$, caz în care următorul scop compus este:
  ?- $\mu(\delta_1), \ldots, \mu(\delta_{i-1}), \mu(\gamma_1), \ldots, \mu(\gamma_q), \mu(\delta_{i+1}), \ldots, \mu(\delta_s)$.
  adică, în scopul compus $\delta_1, \ldots, \delta_s$, se înlocuiește subscopul δ_i, cu **membrul drept al regulii** β :- $\gamma_1, \ldots, \gamma_q$, iar la subscopurile din lista astfel obținută se aplică unificatorul μ.

Procedeul de mai sus este aplicat în manieră **backtracking**. Mai precis, Prologul aplică:

Algoritmul **backward chaining**

Mai jos, **cazul particular** $q = 0$ este cazul aplicării unui **fapt**.
Notăm cu $id_{Var} : Var \to Term(\mathcal{L}_\tau)$ incluziunea canonică: pentru fiecare $v \in Var$, $id_{Var}(v) = v$.

(apelat din programul principal cu următorii parametri: **backwardChaining**(număr inițial de subscopuri date de formule atomice; lista acestor subscopuri inițiale; substituția id_{Var})

backwardChaining(INTEGER s; formule atomice $\delta_1, \ldots, \delta_s$; unificator μ)
  FOR $i = 1$ TO s
    $\delta'_i := \mu(\delta_i)$
  FOR $i = 1$ TO s
    FOR toate clauzele β :- $\gamma_1, \ldots, \gamma_q$ din baza de cunoștințe
      IF ν este un cel mai general unificator pentru δ'_i și β THEN
        $s' := s - 1 + q$
        IF $s' = 0$ THEN RETURN **true**, WRITE soluție: unificatorul $\nu \circ \mu$
        ELSE **backwardChaining**(s'; $\nu(\delta'_1), \ldots, \nu(\delta'_{i-1})$, $\nu(\gamma_1), \ldots, \nu(\gamma_q)$, $\nu(\delta'_{i+1}), \ldots, \nu(\delta'_s)$; $\nu \circ \mu$)
RETURN **false**

Observăm că algoritmul de mai sus constă într–un **backtracking recursiv**.

Dezavantaj: căutarea de mai sus este de tip **depth first**, așadar, de exemplu, dacă inversăm ultimele două clauze din următoarea bază de cunoștințe:

$arc(N, M)$:- M is $N + 1$.
$drum(X, Y)$:- $arc(X, Y)$.
$drum(X, Y)$:- $arc(X, Z)$, $drum(Z, Y)$.

atunci, de exemplu, la interogarea: ?- $drum(1, 3)$, recursia nu se termină.
Cu notațiile de mai sus, amintesc că:

enunțul $\forall x_1 \ldots \forall x_n \, \alpha$ are forma clauzală $\{\alpha\}$,

$$\forall y_1 \ldots \forall y_k \, [(\gamma_1 \wedge \ldots \wedge \gamma_q) \rightarrow \beta] \models\!\!\!\dashv \forall y_1 \ldots \forall y_k \, [\neg (\gamma_1 \wedge \ldots \wedge \gamma_q) \vee \beta]$$
$$\models\!\!\!\dashv \forall y_1 \ldots \forall y_k \, (\neg \gamma_1 \vee \ldots \vee \neg \gamma_q \vee \beta),$$

având forma clauzală $\{\neg \gamma_1, \ldots, \neg \gamma_q, \beta\}$,

$$\neg [\exists z_1 \exists z_2 \ldots \exists z_p \, (\delta_1 \wedge \ldots \wedge \delta_s)] \models\!\!\!\dashv \forall z_1 \forall z_2 \ldots \forall z_p \neg (\delta_1 \wedge \ldots \wedge \delta_s)$$
$$\models\!\!\!\dashv \forall z_1 \forall z_2 \ldots \forall z_p \, (\neg \delta_1 \vee \ldots \vee \neg \delta_s),$$

având forma clauzală $\{\neg \delta_1, \ldots, \neg \delta_s\}$.

Pasul de aplicare a unui **fapt** sau a unei **reguli** descris mai sus din backtracking–ul urmat de Prolog pentru a răspunde interogării ?- $\delta_1, \ldots, \delta_s$ constă în **regula**

aplicată clauzei corespunzătoare negației acestei interogări și clauzei corespunzătoare faptului, respectiv regulii de mai sus, în modul următor: pentru a satisface **subscopul** δ_i pentru un $i \in \overline{1, s}$:

- cu **faptul** α:

$$\frac{\{\neg\,\delta_1, \ldots, \neg\,\delta_{i-1}, \cancel{\neg\,\delta_i}, \neg\,\delta_{i+1}, \ldots, \neg\,\delta_s\},\ \{\cancel{\alpha}\ \}\ \ \text{(cu unificatorul } \mu)}{\{\neg\,\mu(\delta_1), \ldots, \neg\,\mu(\delta_{i-1}), \neg\,\mu(\delta_{i+1}), \ldots, \neg\,\mu(\delta_s)\}}$$

- prin **regula** β :- $\gamma_1, \ldots, \gamma_q$:

$$\frac{\{\neg\,\delta_1, \ldots, \neg\,\delta_{i-1}, \cancel{\neg\,\delta_i}, \neg\,\delta_{i+1}, \ldots, \neg\,\delta_s\},\ \{\neg\,\gamma_1, \ldots, \neg\,\gamma_q, \cancel{\beta}\ \}\ \ \text{(cu unificatorul } \mu)}{\{\neg\,\mu(\delta_1), \ldots, \neg\,\mu(\delta_{i-1}), \neg\,\mu(\gamma_1), \ldots, \neg\,\mu(\gamma_q), \neg\,\mu(\delta_{i+1}), \ldots, \neg\,\mu(\delta_s)\}}$$

Într–adevăr, următorul scop compus pe care trebuie să îl satisfacă Prologul după aplicarea **faptului**, respectiv **regulii** de mai sus este:

$$\mu(\delta_1), \ldots, \mu(\delta_{i-1}), \mu(\delta_{i+1}), \ldots, \mu(\delta_s),$$

respectiv

$$\mu(\delta_1), \ldots, \mu(\delta_{i-1}), \mu(\gamma_1), \ldots, \mu(\gamma_q), \mu(\delta_{i+1}), \ldots, \mu(\delta_s).$$

Toate derivările posibile prin rezoluție respectând strategia de mai sus pot fi reprezentate printr–un **arbore de derivare prin rezoluție SLD**:

- **nodurile** acestui arbore sunt etichetate cu **negațiile scopurilor compuse curente**;
- **rădăcina** arborelui de derivare este etichetată cu **negația scopului compus** din care constă interogarea;
- pentru fiecare nod al arborelui și fiecare clauză din baza de cunoștințe care, împreună cu eticheta acelui nod, are o derivare prin rezoluție, se introduce în arbore un fiu al acelui nod, etichetat cu **negația noului scop compus** obținută prin acea derivare prin rezoluție;
- muchiile arborelui se etichetează cu clauzele din baza de cunoștințe pentru care s–a aplicat regula rezoluției și unificatorii folosiți la fiecare astfel de derivare prin rezoluție.

Conform celor de mai sus:

- dacă măcar una dintre frunze este etichetată cu $\square$, atunci răspunsul la acea interogare este **true**, iar soluția interogării este compunerea unificatorilor de pe drumul de la rădăcină la acea frunză;
- dacă niciuna dintre frunze nu este etichetată cu $\square$, atunci răspunsul la acea interogare este **false**.

Exemplu

Pentru exemplul de mai sus în care am aplicat rezoluția propozițională, avem arborele de derivare prin rezoluție SLD:

Pentru simplitate, putem omite acoladele din etichetele nodurilor arborelui de derivare, ca mai sus.

(**Rezoluția SLD** pentru o **clauză scop** și o **mulțime de clauze Horn** este **corectă și completă**)

Rezoluția SLD aplicată ca mai sus, pornind de la o **clauză scop** și aplicând, la fiecare pas, **regula rezoluției** pentru **clauza scop curentă** și o **clauză Horn**, întotdeauna se termină într–un număr finit de pași și dă răspunsul corect: există o astfel de derivare prin rezoluție a clauzei vide □ ddacă negația acelei clauze scop se deduce semantic din acea mulțime de clauze Horn.

Arborele de derivare prin rezoluție SLD dă **toate soluțiile** unei interogări (care nu sunt neapărat distincte pentru diferitele frunze etichetate cu □), sub forma compunerilor unificatorilor pe fiecare drum de la rădăcina arborelui la o frunză

etichetată cu □, de fapt de substituțiile date prin valorile în variabilele din interogare ale acestor compuneri compuse la stânga cu substituții arbitrare.
Dar Prologul generează acești arbori în manieră top–down, iar generarea unui subarbore al unui nod poate continua la infinit, în timp ce noduri aflate la dreapta acelui nod (în ordinea de generare respectând ordinea în care sunt trecute clauzele în baza de cunoștințe) pot avea subarbori finiți cu noduri frunză etichetate cu clauza vidă □. În unele cazuri, Prologul umple stiva de lucru după eventuala generare a câtorva soluții, fără a obține toate soluțiile.

Exemplu (observați frunzele cărora le corespund aceleași soluții)

Cum obține Prologul soluțiile unei interogări?

Să presupunem că FNC prenex corespunzătoare bazei de cunoștințe este γ.
Fie $n \in \mathbb{N}^*$, și să considerăm n variabile distincte $X_1, \ldots, X_n$.
Atunci, pentru o interogare constând dintr–un scop $\varphi(X_1, \ldots, X_n)$, așadar corespunzând formulei $\exists X_1 \ldots \exists X_n \, \varphi(X_1, \ldots, X_n)$:

• **toate soluțiile** acestei interogări constau în substituțiile $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$, cu $t_1, \ldots, t_n$ termeni, având proprietatea că:

$$\{\gamma\} \models \forall V_1 \ldots \forall V_j \, \sigma(\varphi(X_1, \ldots, X_n)) = \forall V_1 \ldots \exists V_j \, \varphi(t_1, \ldots, t_n),$$

unde $\{V_1, \ldots, V_j\} = V(t_1) \cup \ldots \cup V(t_n)$;

• **soluțiile** *date acestei interogări de către* **Prolog** constau în substituții $\mu_1, \ldots \mu_k$ ($k \in \mathbb{N}$) cu proprietatea că orice soluție a acestei interogări este o compunere între o substituție (arbitrară) și una dintre substituțiile $\mu_1, \ldots \mu_k$, mai precis cu proprietatea că mulțimea tuturor soluțiilor acestei interogări este:
$\{ \{X_1/(\sigma \circ \mu_i)(X_1), \ldots, X_n/(\sigma \circ \mu_i)(X_n)\} \mid i \in \overline{1, k}, \sigma : Var \to Term(\mathcal{L}_\tau)\}$.
Amintesc că fiecare soluție μ_i este dată de compunerea $\nu_i = \nu_{i,1} \circ \nu_{i,2} \ldots \circ \nu_{i,p}$ a unificatorilor de pe drumul de la rădăcină la câte o frunză etichetată cu clauza vidă $\square$ în arborele de derivare prin rezoluție SLD, mai precis de substituția $\mu_i = \{X_1/\nu_i(X_1), \ldots, X_n/\nu_i(X_n)\}$. Putem spune, simplu, că soluțiile date de Prolog sunt valorile substituțiilor $\mu_1, \ldots \mu_k$ în variabilele $X_1, \ldots, X_n$:
$\mu_i(X_1) = \nu_i(X_1), \ldots, \mu_i(X_n) = \nu_i(X_n)$, pentru fiecare $i \in \overline{1, k}$.

Iată și arborele de derivare prin rezoluție SLD pentru interogarea:

?- $concat(Ce, CuCe, [a, b, c])$.

având baza de cunoștințe: $\begin{cases} concat([\,], L, L). \\ concat([H|T], L, [H|M]) \ :- \ concat(T, L, M). \end{cases}$

concat([],L,L). {concat([],L,L)} **clauzele Horn corespunzătoare acestui fapt, respectiv acestei reguli**

concat([H|T],L,[H|M]) :- concat(T,L,M). {¬concat(T,L,M),concat([H|T],L,[H|M])}

?- concat(Ce,CuCe,[a,b,c]).{¬concat(Ce,CuCe,[a,b,c])}

clauza scop corespunzătoare acestei interogări

Etichetăm muchiile arborelui cu clauzele Horn și substituțiile folosite pentru aplicarea regulii rezoluției.

Compunere substituții pe drumul dintre rădăcină și această frunză:
{Ce/[],CuCe/[a,b,c],L/[a,b,c]}
Soluție: {Ce/[],CuCe/[a,b,c]}

Celelalte trei compuneri de substituții, respectiv soluții:

A se vedea în secțiunea *Inițiere în programarea în Prolog* de mai sus rezolvarea acestei interogări în manieră backtracking.