Nagyhatékonyságú logikai programozás

Jegyzetek a BME informatikus hallgatói számára

Szeredi Péter, Benkő Tamás

Számítástudományi
és Információelméleti Tanszék

IQSOFT Rt.

{szeredi,benko}@iqsoft.hu

- A CLP (Constraint Logic Programming) terület áttekintése
- A SICStus `clpq/r` könyvtárai
- A SICStus `clpb` könyvtára
- A SICStus `clpfd` könyvtára
- A SICStus `chr` könyvtára
- A Mercury programozási nyelv

Budapest 2002. szeptember

Nagyhatékonyságú logikai programozás

A tárgy témakörei

- Korlát-logikai programozás (CLP — Constraint Logic Programming)
- A Mercury „nagybani” logikai programozási nyelv

Információk a korlát-logikai programozásról

- „Sárga könyv”: Kim Mariott, Peter J. Stuckey, Programming with Constraints: an Introduction, MIT Press 1998 (részletesebben lásd `http://www.cs.mu.oz.au/~pjs/book/book.html`)
- „Az első alapkönyv”: Pascal Van Hentenryck: Constraint Satisfaction in Logic Programming, MIT Press, 1989
- On-line Guide to Constraint Programming, by Roman Barták (`http://kti.ms.mff.cuni.cz/~bartak/constraints/`)
- Korlát-programozási archívum: `http://www.cs.unh.edu/ccc/archive`

Információk a Mercury nyelvről

- Honlap: `http://www.cs.mu.oz.au/research/mercury/`

A CLP alapgondolata

A CLP($\mathcal{X}$) séma

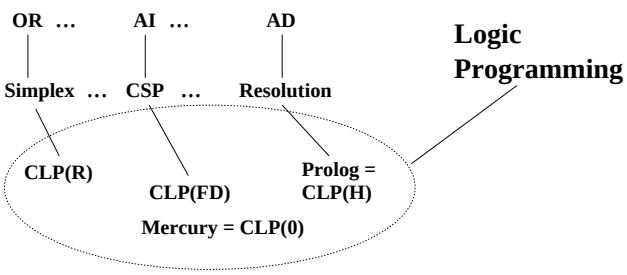
Prolog + egy valamilyen $\mathcal{X}$ adattartományra és azon értelmezett korlátokra (relációkra) vonatkozó „erős” következtetési mechanizmus.

Példák az $\mathcal{X}$ tartomány megválasztására

$\mathcal{X}$ = Q vagy R (a racionális vagy valós számok)
korlátok = lineáris egyenlőségek és egyenlőtlenségek
következtetési mechanizmus = Gauß elimináció és szimplex módszer

$\mathcal{X}$ = FD (egész számok Véges Tartománya, angolul FD — Finite Domain)
korlátok = különféle aritmetikai és kombinatorikus relációk
következtetési mechanizmus = MI CSP–módszerek (CSP = Korlát-Kielégítési Probléma)

$\mathcal{X}$ = B (0 és 1 Boole értékek)
korlátok = ítéletkalkulusbeli relációk
következtetési mechanizmus = MI SAT-módszerek (SAT — Boole kielégíthetőség)



Példa: CLP(MiniNat)

Egy miniatűr kvázi-CLP nyelv természetes számokra

- Tartomány: Nem negatív egészek
- Függvények:

  `+ - *`

- Korlát-relációk:

  `= < > =< >=`

- Korlát-megoldó algoritmus:

  SICStus korutin-kiterjesztésén alapul

A Prologba ágyazás szintaxisa:

{*Korlát*} a *Korlát* felvétele
(`{X}` szintaktikus édesítőszer, ekvivalens a `'{}'(X)` kifejezéssel.)

Példafutás

```
| ?- {2*X+3*Y=8}.
X = 1, Y = 2 ? ;
X = 4, Y = 0 ? ;
no
| ?- {X*2+1=28}.
no
| ?- {X*X+Y*Y=25, X > Y}.
X = 4, Y = 3 ? ;
X = 5, Y = 0 ? ;
no
```

Prolog háttér: blokkolás, korutinszervezés

Blokk-deklarációk SICStusban
Egy eljárásra előírhatjuk, hogy mindaddig, amíg egy ún. blokkolási feltétel fennáll, az eljárás függesztődjék fel. Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesítetlen változó (blokkolási feltétel), akkor a `p` hívás felfüggesztődik.

Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.
`:- block p(-, ?), p(?, -).`

Blokk-deklarációk haszna

- Adatfolyam-programozás (lásd Hamming probléma, Prolog jegyzet)
- Generál és ellenőriz programok gyorsítása
- Végtelen választási pontok kiküszöbölése

Biztonságos `append/3`, blokk-deklarációval

```
:- block append(-, ?, -).
% blokkol, ha az első és a harmadik argumentum
% egyaránt behelyettesítetlen
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```


Példa korutinszervezésre: többirányú összeadás

```
% X+Y=Z, ahol X, Y és Z természetes számok.
% Bármelyik argumentum lehet behelyettesítetlen.
plusz(X, Y, Z) :-
        append(A, B, C),
        len(A, X),
        len(B, Y),
        len(C, Z).

% L hossza Len.
len(L, Len) :-
        len(L, 0, Len).

:- block len(-, ?, -).
% L lista hossza Len-Len0. Len0 mindig ismert.
len(L, Len0, Len) :-
        nonvar(Len), !, Len1 is Len-Len0,
        length(L, Len1).
len(L, Len0, Len) :-
        % nonvar(L), % a blokkolási feltétel miatt!
        (   L == [] -> Len = Len0
        ;   L = [_|L1],
            Len1 is Len0+1, len(L1, Len1, Len)
        ).
```

```
| ?- plusz(X, Y, 2).
X = 0, Y = 2 ? ;
X = 1, Y = 1 ? ;
X = 2, Y = 0 ? ;
no
| ?- plusz(X, X, 8).
X = 4 ? ;
no
| ?- plusz(X, 1, Y), plusz(X, Y, 20).
no
```


További korutinszervező eljárások

Hívások késleltetése

`freeze(X, Hivas)`
`Hivas`t felfüggeszti mindaddig, amíg `X` behelyettesítetlen változó.

`dif(X, Y)`
`X` és `Y` nem egyesíthető. Mindaddig felfüggesztődik, amíg ez el nem dönthető.

`when(Feltétel, Hívás)`
Blokkolja a `Hívás`t mindaddig, amíg a `Feltétel` igazzá nem válik. Itt a `Feltétel` egy (nagyon) leegyszerűsített Prolog cél, amelynek szintaxisa:

```
CONDITION ::=  nonvar(X) | ground(X) | ?=(X,Y) |
               CONDITION, CONDITION |
               CONDITION; CONDITION
```

(`ground(X)` jelentése: `X`, tömör, azaz nem tartalmaz (behelyettesítetlen) változót
`?=(X,Y)` jelentése: `X` és `Y` egyesíthetősége eldönthető.)
Példa:

```
| ?- when( ((nonvar(X);?=(X,Y)),ground(T)),
           process(X,Y,T)).
```

Késleltetett hívások lekérdezése

`frozen(X, Hivas)`
Az `X` változó miatt felfüggesztett hívás(oka)t egyesíti `Hivas`-sal.

`call_residue(Hivas, Maradék)`
`Hivas`-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja `Maradékban`. Pl.

```
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).
X = f(Z),
Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))] ?
```


CLP(MiniNat) megvalósítása

Számábrázolás

- A korábbi `plusz/3` eljárásban egy N elemű listával ábrázoltuk az N számot (a listaelemek érdektelenek, általában behelyettesítetlen változók)
- Példa: a `2` szám ábrázolása: `[_,_]` $\equiv$ `.(_,.(_,[]))`.
- Hagyjuk el a felesleges változókat, akkor a `2` szám ábrázolása: `.(.([]))`.
- Itt a `[]` jelenti a 0 számot, a `.(X)` struktúra az `X` szám rákövetkezőjét (a nála 1-gyel nagyobb számot).
- Ez tulajdonképpen a Peano féle számábrázolás, ha a `. /1` helyett az `s/1` funktort, a `[]` helyett a `0` konstanst használjuk.
- A CLP(MiniNat) megvalósításában a Peano számábrázolást használjuk, tehát; `0 = 0; 1 = s(0); 3 = s(s(s(0)))` stb.

Összeadás és kivonás

```
% plusz(X, Y, Z): X+Y=Z (Peano számokkal).
:- block plusz(-, ?, -).
plusz(0, Y, Y).
plusz(s(X), Y, s(Z)) :-
        plusz(X, Y, Z).

% +(X, Y, Z): X+Y=Z (Peano számokkal). Hatékonyabb, mert
% továbblép, ha bármelyik argumentum behelyettesített.
:- block +(-, -, -).
+(X, Y, Z) :-
        var(X), !, plusz(Y, X, Z).  % \+((var(Y),var(Z)))
+(X, Y, Z) :-
        /* nonvar(X), */ plusz(X, Y, Z).

% X-Y=Z (Peano számokkal).
-(X, Y, Z) :-
        +(Y, Z, X).
```

CLP(MiniNat) megvalósítása (folyt.)

A szorzás művelet megvalósítási elvei:

- Felfüggesztjük mindaddig, míg legalább egy tényező vagy a szorzat ismertté nem válik.
- Ha az egyik tényező ismert, visszavezetjük ismételt összeadásra.
- Ha a szorzat ismert (N), az egyik tényezőre végigpróbáljuk az 1,2,... N értékeket, ezáltal ismételt összeadásra visszavezethetővé tesszük.

```
% X*Y=Z. Blokkol, ha nincs tömör argumentuma.
*(X, Y, Z) :-
        when( (ground(X);ground(Y);ground(Z)),
              szorzat(X, Y, Z)).

% X*Y=Z, ahol legalább az egyik argumentum tömör.
szorzat(X, Y, Z) :-
        (   ground(X) -> szor(X, Y, Z)
        ;   ground(Y) -> szor(Y, X, Z)
        ;   /* Z tömör! */
            Z == 0 -> szorzatuk_nulla(X, Y)
        ;   +(X, _, Z), % X =< Z, vö. between(1, Z, X)
            szor(X, Y, Z)
        ).

% X*Y=0.
szorzatuk_nulla(X, Y) :-
        ( X = 0 ; Y = 0 ).

% szor(X, Y, Z): X*Y=Z, X tömör.
% Y-nak az (ismert) X-szeres összeadása adja ki Z-t.
szor(0, _X, 0).
szor(s(X), Y, Z) :-
        +(Z1, Y, Z),
        szor(X, Y, Z1).
```


CLP(MiniNat) megvalósítása: (folyt. 2)

A korlátok végrehajtása

- A funkcionális alakban megadott korlátokat a `+ /3, - /3, * /3` hívásokból álló célsorozattá alakítjuk, majd ezt a célsorozatot meghívjuk.
- Például a `{X*Y+2=Z}` korlát lefordított alakja:
  `*(X, Y, _A),+(_A, s(s(0)), Z)`,
- Az `{X =< Y}` korlátot az `{X+_ = Y}` korlátra, az `{X < Y}` korlátot pedig az `{X+s(_) = Y}` korlátra vezetjük vissza

```
% {Korlat}: Korlat fennáll.
{Korlat} :-
        korlat_cel(Korlat, Cel), call(Cel).
```

Korlátok fordítása

```
% korlat_cel(Korlat, Cel): Korlat végrehajtható
% alakja a Cel célsorozat.
korlat_cel(Kif1=Kif2, (C1,C2)) :-
        kiertekel(Kif1, E, C1),
        kiertekel(Kif2, E, C2).
korlat_cel(Kif1 =< Kif2, Cel) :-
        korlat_cel(Kif1+_ = Kif2, Cel).
korlat_cel(Kif1 < Kif2, Cel) :-
        korlat_cel(s(Kif1) =< Kif2, Cel).
korlat_cel(Kif1 >= Kif2, Cel) :-
        korlat_cel(Kif2 =< Kif1, Cel).
korlat_cel(Kif1 > Kif2, Cel) :-
        korlat_cel(Kif2 < Kif1, Cel).
korlat_cel((K1,K2), (C1,C2)) :-
        korlat_cel(K1, C1),
        korlat_cel(K2, C2).
```


CLP(MiniNat) megvalósítása: (folyt. 3)

Kifejezések fordítása

- Egy `Kif1` *Op* `Kif2` kifejezés lefordított alakja egy három részből álló célsorozat, amely egy `E` változóban állítja elő a kifejezés eredményét:
  - első rész: `Kif1` értékét pl. `A`-ban előállító cél(sororzat).
  - második rész: `Kif2` értékét pl. `B`-ban előállító cél(sororzat).
  - harmadik rész: az *Op*`(A, B, E)` hívás (ahol *Op* a `+, -, *` jelek egyike).
- Egy szám lefordított formája az ő Peano alakja.
- Minden egyéb (változó, vagy már Peano alakú szám) változatlan marad a fordításkor.

```
% kiertekel(Kif, E, Cel): A Kif aritmetikai kifejezés
% értékét E-ben előállító cél Cel.
% Kif egészekből a +, -, és * operátorokkal épül fel.
kiertekel(Kif, E, (C1,C2,Rel)) :-
        nonvar(Kif),
        Kif =.. [Op,Kif1,Kif2], !,
        kiertekel(Kif1, E1, C1),
        kiertekel(Kif2, E2, C2),
        Rel =.. [Op,E1,E2,E].
kiertekel(N, Kif, true) :-
        number(N), !,
        int_to_peano(N, Kif).
kiertekel(Kif, Kif, true).

% int_to_peano(N, P): N természetes szám Peano alakja P.
int_to_peano(0, 0).
int_to_peano(N, s(P)) :-
        N > 0, N1 is N-1,
        int_to_peano(N1, P).
```


Prolog háttér: kifejezések testreszabott kiírása

`print/1`
Alapértelmezésben azonos `write`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó részkifejezésre meghívja `portray`-t. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghiúsulás esetén maga írja ki a részkifejezést.
A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására!

`portray/1`
Igaz, ha `Kif` kifejezést a Prolog rendszernek nem kell kiírnia. Alkalmas formában kiírja a `Kif` kifejezést.
Ez egy felhasználó által definiálandó (*kampó*) eljárás (hook predicate).

```
% Peano számok kiírásának formázása
user:portray(Peano) :-
        peano_to_int(Peano, 0, N), write(N).

% A Peano Peano-szám értéke N-N0.
peano_to_int(Peano, N0, N) :-
        nonvar(Peano),
        (   Peano == 0 -> N = N0
        ;   Peano = s(P),
            N1 is N0+1,
            peano_to_int(P, N1, N)
        ).

% felfüggesztett célok kiíratásának formázása
user:portray(user:Rel) :-
        Rel =.. [Op,A,B,C],
        (   Op = (+) ; Op = (-) ; Op = (*) ),
        Fun =.. [Op,A,B],
        print({Fun=C}).
```

CLP(MiniNat) használata — példa

```
:- block fact(-,-).
fact(N, F) :-
        {N = 0, F = 1}.
fact(N, F) :-
        {N >= 1, N1 = N-1},
        fact(N1, F1),
        {F = N*F1}.

| ?- fact(6, F).
F = 720 ? ; no

| ?- fact(8, F).
F = 40320 ? ; no

| ?- fact(F, 6).
F = 3 ? ; no

| ?- fact(F, 24).
F = 4 ? ;
! Resource error: insufficient memory

| ?- fact(F, 10).
no

| ?- fact(F, 11).
! Resource error: insufficient memory

| ?- {X*X+Y*Y=25, X>Y}.
X = 4, Y = 3 ? ;
X = 5, Y = 0 ? ;
X = 5, Y = 0 ? ;
no
```


Prolog háttér: programok előfeldolgozása

Kampó eljárások a fordítási idejű átalakításhoz:

- `term_expansion(+Kif, -Klózok)`: Minden betöltő eljárás (`consult, compile` stb.) által beolvasott kifejezésre a rendszer meghívja. A második, kimenő paraméterben várja a transzformált alakot (lehet lista is). Meghiúsulás esetén változtatás nélkül veszi fel a kifejezést klózként.
- `goal_expansion(+Cél, +Modul, -ÚjCél)`: Minden a beolvasott programban (vagy feltett kérdésben) előforduló részcélra meghívja a rendszer. A harmadik, kimenő paraméterben várja a transzformált alakot (lehet konjunkció). Meghiúsulás esetén változtatás nélkül hagyja a célt.

CLP(MiniNat) továbbfejlesztése `goal_expansion` használatával

- A funkcionális alak átalakítása a betöltés alatt is elvégezhető:

```
user:goal_expansion({Korlat}, _, Cel) :-
        korlat_cel(Korlat, Cel).
```

- A faktoriális példa betöltött alakja (a `true` hívások elhagyása után):

```
fact(0, s(0)).
fact(N, F) :-
        +(s(0), _, N),   % N >= 1
        -(N, s(0), N1),  % N1 = N-1
        *(N, F1, F),     % F = N*F1
        fact(N1, F1).
```

- Vigyázat! Az így előálló kód már nem foglalkozik a számok Peano-alakra hozásával:

```
| ?- fact(N, 120).         -->  no
| ?- {F=120}, fact(N, F).  -->  F = 120, N = 5 ? ; no
```

Megjegyzés: a faktoriális példában nincs mérhető gyorsulás

CLP(MiniNat) javított változatai

A nulla szorzat problémája

```
| ?- {X*X=0}.
X = 0 ? ; X = 0 ? ; no
```

A probléma 1. javítása

```
% X*Y=0, ahol X és Y Peano számok.
szorzatuk_nulla(X, Y) :-
        (   X = 0
        ;   X \== Y, Y = 0
        ).

| ?- {X*X=0}.
X = 0 ? ; no

| ?- {X*Y=0}, X=Y.
X = 0, Y = 0 ? ;
X = 0, Y = 0 ? ; no
```

A probléma 2. javítása

```
% X*Y=0, ahol X és Y Peano számok.
szorzatuk_nulla(X, Y) :-
        (   X = 0
        ;   dif(X, 0), Y = 0
        ).

| ?- {X*Y=0}, X=Y.
X = 0, Y = 0 ? ; no
```


CLP(MiniNat) javított változatai (folyt)

Az erőforrás probléma

- A `fact(N, 11)` hívás a második klózzal illesztve a `{11=N*F1}` feltételre vezetődik vissza. Ez két megoldást generál (`N=1,F1=11`, ill. `N=11,F1=1`. Ezekre a behelyettesítésekre felébred a rekurzív `fact` hívás először a `fact(0,11)` majd a `fact(10,1)` paraméterekkel.
- A `fact/2` második klóza ez utóbbit mohón értékeli ki: kiszámolná `10!`-t, és csak ezután egyesítené `1`-gyel. Azonban a `10!` kiszámolásához (Peano számként) kevés a memória `:-(`.
- A probléma javítása: a szorzat-feltételt tegyük a rekurzív `fact/2` hívás elé.

```
:- block fact(-,-).
fact(N, F) :- {N = 0, F = 1}.
fact(N, F) :-
        {N >= 1, N1 = N-1, F = N*F1},
        fact(N1, F1).

| ?- fact(N, 24).    -------->   N = 4 ? ; no
```

- Azonban az alábbi cél futása még így is kivárhatatlan …

```
| ?- fact(N, 720).   -------->   N = 6 ? ;
```

Megjegyzések

- Egy korlát-programban minél később célszerű választási pontot csinálni.
- Ideálisan csak az összes korlát felvétele után kezdjük meg a keresést.
- Megoldás: egy külön keresési fázis (az ún. címkézés, labeling):

```
program :-
      korlátok_felvétele(...), labeling([V1, ..., VN]).
```

- CLP(MiniNat)-ban az ismertetett eszközökkel ez nehezen megoldható, de
- CLP(MiniB) esetén (lásd 1. kis házi feladat) könnyen készíthető ilyen `labeling/1` eljárás.

1. kis házi feladat: CLP(MiniB) megvalósítása

CLP(MiniB) jellemzése

- **Tartomány:** logikai értékek (`1` és `0`, igaz és hamis)
- **Függvények** (egyben korlát-relációk):

`~ P`	`P` hamis (*negáció*).
`P * Q`	`P` és `Q` mindegyike igaz (*konjunkció*).
`P + Q`	`P` és `Q` legalább egyike igaz (*diszjunkció*).
`P # Q`	`P` és `Q` pontosan egyike igaz (*kizáró vagy*).
`P =\= Q`	Ugyanaz mint `P # Q`.
`P =:= Q`	Ugyanaz mint `~(P # Q)`.

A megvalósítandó eljárások

- `sat(`*Kif*`)`, ahol *Kif* változókból, a `0`, `1` konstansokból a fenti müveletekkel felépített logikai kifejezés. Jelentése: A *Kif* logikai kifejezés igaz. A `sat/1` eljárás ne hozzon létre választási pontot! A benne szereplő változók behelyettesítése esetén minél előbb ébredjen fel, és végezze el a megfelelő következtetéseket (lásd a példákat alább)!
- `count(`*Es*`, `*N*`)`, ahol *Es* egy (változó-)lista, *N* adott természetes szám. Jelentése: Az *Es* listában pontosan *N* olyan elem van, amelynek értéke 1.
- `labeling(`*Változók*`)`. Behelyettesíti a *Változók*at `0`, `1` értekekre. Visszalépés esetén felsorolja az összes lehetséges értéket.

Futási példák

```
| ?- sat(A*B =:= (~A)+B).
                        --->   <...felfüggesztett célok...> ? ; no
| ?- sat(A*B =:= (~A)+B), labeling([A,B]).
                        --->   A = 1, B = 0 ? ; A = 1, B = 1 ? ; no
| ?- sat((A+B)*C=\=A*C+B), sat(A*B).
                        --->   A = 1, B = 1, C = 0 ? ; no
| ?- count([A,A,B], 2).  --->   <...felfüggesztett célok...> ? ; no
| ?- count([A,A,B], 2), labeling([A]).
                        --->   A = 1, B = 0 ? ; no
| ?- count([A,A,B,B], 3), labeling([A,B]).
                        --->   no
| ?- sat(~A =:= A).      --->   no
```


1. kis házi feladat: egy kis segítség

```
:- op(100, fx, ~).

~(A, B) :-
        when( (nonvar(A); nonvar(B); ?=(A,B)),
              not(A,B)
            ).

not(A, NA) :-
        (   nonvar(A) -> NA is 1-A
        ;   nonvar(NA) -> A is 1-NA
        ;   A == NA -> fail
        ).
```

```
| ?- trace, ~(A, A).
 1 1 Call: ~(A,A) ?
 2 2 Call: when((nonvar(A);nonvar(A);?=(A,A)),not(A,A)) ?
 3 3 Call: not(A,A) ?
 4 4 Call: nonvar(A) ?
 4 4 Fail: nonvar(A) ?
 5 4 Call: nonvar(A) ?
 5 4 Fail: nonvar(A) ?
 6 4 Call: A==A ?
 6 4 Exit: A==A ?
 3 3 Fail: not(A,A) ?
 2 2 Fail: when((nonvar(A);nonvar(A);?=(A,A)),not(A,A)) ?
 1 1 Fail: ~(A,A) ?
no
| ?- sat(A*A=:=B).
                                B = A ? ; no
| ?- sat(A#A=:=B).
                                B = 0 ? ; no
| ?- sat(A+B=:=C), A=B.
                                B = A, C = A ? ; no
```


CLP rendszerek a nagyvilágban

Néhány implementáció

- clp(R) — az első CLP(X) rendszer (Monash Univ, Australia, IBM Yorktown Heights és CMU)
- CHIP — FD, Q és B (ECRC, München, Cosytec, Franciao.); CHARME (Bull); Decision Power (ICL)
- Prolog III, Prolog IV (PrologIA, Marseille), Q (nem-lineáris is), B, FD, listák, intervallumok
- ILOG solver (ILOG, Franciao.) — C++ könyvtár: R (nem-lineáris is), FD, halmazok
- SICStus Prolog (SICS, Svédo.) — R/Q, FD, B, CHR
- GNU Prolog (INRIA, Franciao.) — FD (C-re fordít)
- Oz (DFKI, Németo.) — korlát alapú elosztott funkcionális nyelv.

Kommerciális rendszerek (a fentiek között)

- ILOG, CHIP, Prolog III–IV, SICStus
- a szakma óriása: ILOG
  - szakterület: CLP + vizualizációs eszközök + szabályalapú eszközök
  - felvásárolta az egyik vezető operációkutatási céget, a CPLEX-et
  - 400 munkatárs 7 országban
  - 55M USD éves bevétel
  - NASDAQ-on jegyzett

Mire használják a CLP rendszereket

Ipari erőforrás optimalizálás

- termék- és gépkonfiguráció
- gyártásütemezés
- emberi erőforrások ütemezése
- logisztikai tervezés

Közlekedés, szállítás

- repülőtéri allokációs feladatok (beszállókapu, poggyász-szalag stb.)
- repülő-személyzet járatokhoz rendelése
- menetrendkészítés
- forgalomtervezés

Távközlés, elektronika

- GSM átjátszók frekvencia-kiosztása
- lokális mobiltelefon-hálózat tervezése
- áramkörtervezés és verifikálás

Egyéb

- szabászati alkalmazások
- grafikus megjelenítés megtervezése
- multimédia szinkronizáció
- légifelvételek elemzése

A CLP($\mathcal{X}$) séma

Egy adott CLP($\mathcal{X}$) meghatározásakor meg kell adni

- a korlát-következtetés tartományát,
- a korlátok szintaxisát és jelentését (függvények, relációk),
- a korlát-megoldó algoritmust.

A korlátok osztályozása

- *egyszerű korlátok* — a korlát-megoldó azonnal tudja kezelni őket;
- *összetett korlátok* — felfüggesztve, démonként várnak arra, hogy a korlát-megoldónak segíthessenek.

A CLP($\mathcal{X}$) korlát-megoldók közös vonása: a *korlát tár*

- A korlát tár *konzisztens* korlátok halmaza (konjunkciója).
- A korlát tár elemei egyszerű korlátok.
- A közönséges Prolog végrehajtás során a kurrens célsorozat mellett a CLP($\mathcal{X}$) rendszer nyilvántartja a korlát tár állapotát:
  - amikor a végrehajtás egy egyszerű korláthoz ér, akkor azt a megoldó megpróbálja hozzávenni a tárhoz;
  - ha az új korlát hozzávételével a tár konzisztens marad, akkor ez a redukciós lépés sikeres és a tár kibővül az új korláttal;
  - ha az új korlát hozzávételével a tár inkonzisztenssé válna, akkor (nem kerül be a tárba és) meghiúsulást, azaz visszalépést okoz;
  - visszalépés esetén a korlát tár is visszaáll a korábbi állapotába.
- a összetett korlátok démonként (ágensként) várakoznak arra, hogy:
  a. egyszerű korláttá váljanak
  b. a tárat egy egyszerű következményükkel bővíthessék (az ún. erősítés)

A SICStus clp(Q,R) könyvtárak

A `clpq`/`clpr` könyvtárak

- Tartomány:
  - `clpr`: lebegőpontos számok
  - `clpq`: racionális számok
- Függvények:
  `+ - * / min max pow exp` (kétargumentumúak, `pow` ≡ `exp`),
  `+ - abs sin cos tan` (egyargumentumúak).
- Korlát-relációk:
  `= =:= < > =< >= =\=` (`=` ≡ `=:=`)
- Primitív korlátok (korlát tár elemei):
  lineáris kifejezéseket tartalmazó relációk
- Korlát-megoldó algoritmus:
  lineáris programozási módszerek: Gauss eliminació, szimplex módszer

A könyvtár betöltése:

- `use_module(library(clpq))`, vagy
- `use_module(library(clpr))`

A fő beépített eljárás

- `{` *Korlát* `}` , ahol *Korlát* változókból és (egész vagy lebegőpontos) számokból a fenti műveletekkel felépített reláció, vagy ilyen relációknak a vessző (`,`) operátorral képzett konjunkciója.

Példafutás a SICStus `clpq` könyvtárával

Példafutás

```
| ?- use_module(library(clpq)).
{loading .../library/clpq.ql...}
...

| ?- {X=Y+4, Y=Z-1, Z=2*X-9}.
X = 6, Y = 2, Z = 3 ?    % lineáris egyenlet

| ?- {X+Y+9<4*Z, 2*X=Y+2, 2*X+4*Z=36}.
                         % lineáris egyenlőtlenség
{X<29/5}, {Y= -2+2*X}, {Z=9-1/2*X} ?
                         % az eredmény: a tár állapota

| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.
{X=100-2*Y} ?            % lineárissá egyszerűsíthető

| ?- {(Y+X)*(X+Y) = Y*Y+100*X}.
                         % így már nem lineáris
clpq:{2*(X*Y)-100*X+X^2=0} ?
                         % a clpq modul-prefix jelzi, hogy
                         % felfüggesztett összetett
                         % hívásról van szó

| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.
                         % nem lineáris...
clpq:{1+2*X+2*(Y*X)-2*X^2+2*Y=0} ?

| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.
X = -1/4, Y = -1/4 ?     % így már igen...

| ?- {2 = exp(8, X)}.     % nem-lineárisak is
                         % megoldhatók
X = 1/3 ?
```


Összetett korlátok kezelése CLP(Q)-ban

Példa várakozó ágensre

```
| ?- {X =< Y}, {X*(Y+1) > X*X+Z},
       (   Z = X*(Y-X), {Y < 0}
       ;   Y = X
       ).
                         Y = X,  {X-Z>0} ? ; no
```

A végrehajtás lépései

```
| ?- {X =< Y}, {X*(Y+1) > X*X+Z}.
                         {X-Y=<0}, clpq:{Z-X-Y*X+X^2<0} ?

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Z = X*(Y-X).
                         Z = X*(Y-X), {X-Y=<0}, {X>0} ?

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Z = X*(Y-X), {Y < 0}.
                         no

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Y = X.
                         Y = X, {X-Z>0} ?
```

Példa egy *lehetséges* **erősítési lépésre**

- A tár tartalma: `X > 3`.
- A végrehajtandó összetett korlát: `Y > X*X`.
- A korlátot a CLP megoldó nem tudja felvenni a tárba, de egy *következményét*, pl. az `Y > 9` korlátot felvehetné!
- Az erősítés után az eredeti összetett korlát továbbra is démonként kell lebegjen!
- **Fontos megjegyzés:** a CLP(Q/R) rendszer **nem** hajtja végre a fenti következtetést, és általánosan semmiféle erősítést nem végez.

Egy összetettebb példa: hiteltörlesztés

```
% Hiteltörlesztés számítása: P összegű hitelt
% Time hónapon át évi IntRate kamat mellett havi MP
% részletekben törlesztve Bal a maradványösszeg.
mortgage(P, Time, IntRate, Bal, MP):-
     {Time > 0, Time =< 1,
     Bal = P*(1+Time*IntRate/1200)-Time*MP}.
mortgage(P, Time, IntRate, Bal, MP):-
     {Time > 1},
     mortgage(P*(1+IntRate/1200)-MP,
              Time-1, IntRate, Bal, MP).


| ?- mortgage(100000,180,12,0,MP).
                        % 100000 Ft hitelt 180
                        % hónap alatt törleszt 12%-os
                        % kamatra, mi a havi részlet?
MP = 1200.1681 ?

| ?- mortgage(P,180,12,0,1200).
                        % ugyanez visszafelé
P = 99985.9968 ?

| ?- mortgage(100000,Time,12,0,1300).
                        % 1300 Ft a törlesztőrészlet,
                        % mi a törlesztési idő?
Time = 147.3645 ?

| ?- mortgage(P,180,12,Bal,MP).

{MP=0.0120*P-0.0020*Bal} ?

| ?- mortgage(P,180,12,Bal,MP), ordering([P,Bal,MP]).

{P=0.1668*Bal+83.3217*MP} ?
```


További könyvtári eljárások

`entailed(Korlát)` — `Korlát` levezethető a jelenlegi tárból.

`inf(Kif, Inf)` ill. `sup(Kif, Sup)` — kiszámolja `Kif` infímumát ill. szuprémumát, és egyesíti `Inf`-fel ill. `Sup`-pal. Példa:

```
| ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15,
       Z = 30*X+50*Y
     }, sup(Z, Sup).

Sup = 310, {....}
```

`minimize(Kif)` ill. `maximize(Kif)` — kiszámolja `Kif` infimumát ill. szuprémumát, és egyenlővé teszi `Kif`-fel. Példa:

```
| ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15,
       Z = 30*X+50*Y
     }, maximize(Z).

X = 7, Y = 2, Z = 310
```

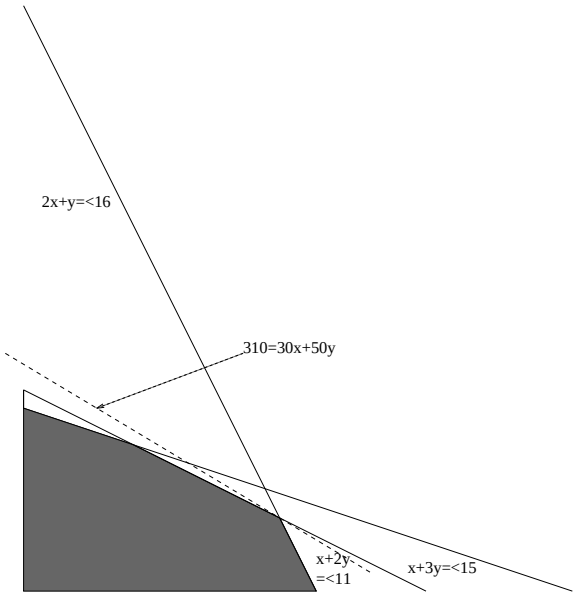
`bb_inf(Egészek, Kif, Inf)` — kiszámolja `Kif` infimumát, azzal a további feltétellel, hogy az Egészek listában levő minden változó egész (ún. „Mixed Integer Optimisation Problem”).

```
| ?- {X >= 0.5, Y >= 0.5}, inf(X+Y, I).

I = 1, {Y>=1/2}, {X>=1/2} ?

| ?- {X >= 0.5, Y >= 0.5}, bb_inf([X,Y], X+Y, I).

I = 2, {X>=1/2}, {Y>=1/2} ?
```

`ordering(V1 < V2)` — A `V1` változó előbb szerepeljen az eredmény-korlátban mint a `V2` változó.

`ordering([V1,V2,...])` — `V1, ...` ebben a sorrendben szerepeljen az eredmény-korlátban.

Szélsőérték-számítás grafikus illusztrálása




```
| ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15,
       Z = 30*X+50*Y
     }, sup(Z, Sup).

Sup = 310, {Z=30*X+50*Y},
           {X+1/2*Y=<8}, {X+3*Y=<15}, {X+2*Y=<11}
```


További részletek

Projekció

```
% Az (X,Y) pont az (1,2) (1,4) (2,4) pontok
% által kifeszített háromszögben van.
hszogben(X,Y) :-
        { X=1*L1+1*L2+2*L3,
          Y=2*L1+4*L2+4*L3,
          L1+L2+L3=1, L1>=0, L2>=0, L3>=0 }.

| ?- hszogben(X,Y).
                     {Y=<4}, {X>=1}, {X-1/2*Y=<0} ?

| ?- hszogben(_, Y).
                     {Y=<4}, {Y>=2} ?

| ?- hszogben(X, _).
                     {X>=1}, {X=<2} ?
```

Belső ábrázolás

`clpr` — lebegőpontos szám; `clpq` — `rat(`*Számláló*`, `*Nevező*`)`, ahol *Számláló* és *Nevező* relatív prímek. Például `clpq`-ban:

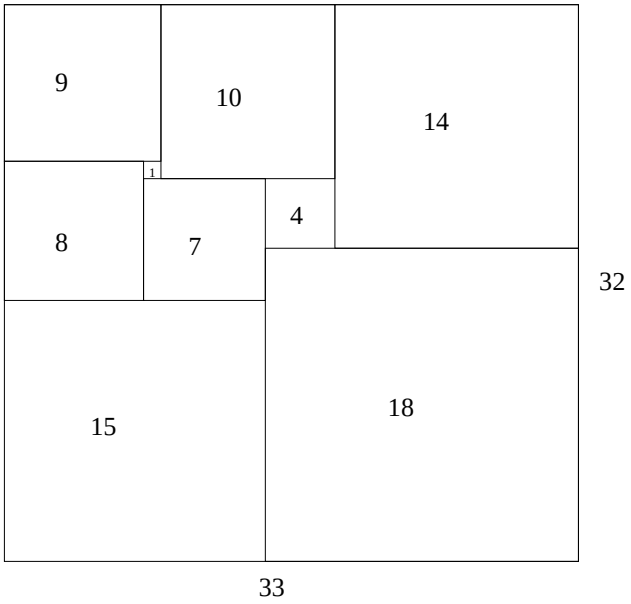
```
| ?- {X=0.5}, X=0.5.
no
| ?- {X=0.5}, X=1/2.
no
| ?- {X=0.5}, X=rat(2,4).
no
| ?- {X=0.5}, X=rat(1,2).
X = 1/2 ?
| ?- {X=5}, X=5.
no
| ?- {X=5}, X=rat(5,1).
X = 5 ?
```

Egy nagyobb CLP(Q) feladat: Tökéletes téglalapok

A feladat

- egy olyan téglalap keresése
- amely kirakható páronként különböző oldalú négyzetekből

Egy megoldás
(a legkevesebb, 9 darab négyzet felhasználásával)



Tökéletes téglalapok — CLP(Q) megoldás

```
% Colmerauer A.: An Introduction to Prolog III,
% Communications of the ACM, 33(7), 69-90, 1990.

% Rectangle 1 x Width is covered by distinct
% squares with sizes Ss.
filled_rectangle(Width, Ss) :-
       { Width >= 1 }, distinct_squares(Ss),
       filled_hole([-1,Width,1], _, Ss, []).

% distinct_squares(Ss): All elements of Ss are distinct.
distinct_squares([]).
distinct_squares([S|Ss]) :-
       { S > 0 }, outof(Ss, S), distinct_squares(Ss).

outof([],     _).
outof([S|Ss], S0) :- { S =\= S0 }, outof(Ss, S0).

% filled_hole(L0, L, Ss0, Ss): Hole in line L0
% filled with squares Ss0-Ss (diff list) gives line L.
% Def: h(L): sum of lengths of vertical segments in L.
% Pre: All elements of L0 except the first >= 0.
% Post: All elems in L >=0, h(L0) = h(L).
filled_hole(L, L, Ss, Ss) :-
       L = [V|_], {V >= 0}.
filled_hole([V|HL], L, [S|Ss0], Ss) :-
       { V < 0 }, placed_square(S, HL, L1),
       filled_hole(L1, L2, Ss0, Ss1), { V1=V+S },
       filled_hole([V1,S|L2], L, Ss1, Ss).

% placed_square(S, HL, L): placing a square size S on
% horizontal line HL gives (vertical) line L.
% Pre: all elems in HL >=0
% Post: all in L except first >=0, h(L) = h(HL)-S.
placed_square(S, [H,V,H1|L], L1) :-
       { S > H, V=0, H2=H+H1 },
       placed_square(S, [H2|L], L1).
placed_square(S, [S,V|L], [X|L]) :- { X=V-S }.
placed_square(S, [H|L], [X,Y|L]) :-
       { S < H, X= -S, Y=H-S }.
```


Tökéletes téglalapok: példafutás

```
% 600 MHz Pentium III
| ?-    length(Ss, N), N > 1, statistics(runtime, _),
        filled_rectangle(Width, Ss),
        statistics(runtime, [_,MSec]).

N = 9, MSec = 8010, Width = 33/32,
Ss = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

N = 9, MSec = 1010, Width = 69/61,
Ss = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61] ? ;

N = 9, MSec = 10930, Width = 33/32,
Ss = [9/16,15/32,7/32,1/4,7/16,1/8,5/16,1/32,9/32] ?
```

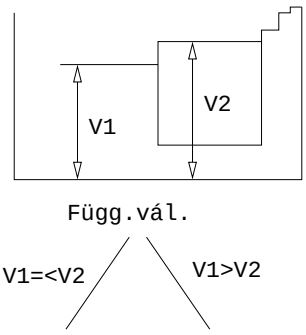
Az `outof` hívás kihagyásával végzett futtatás

Kommentként közöljük az adott ágon generált korlátokat, a redundánsak elhagyásával.

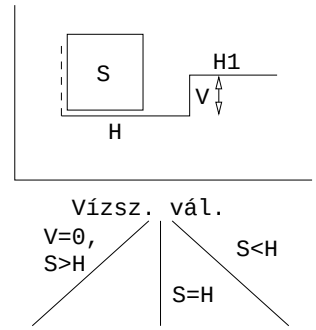
```
| ?- filled_rectangle(W, [S1,S2,S3], [eqsq]).

S1 = 1/2, S2 = 1, S3 = 1/2, W = 3/2 ? ;    % 3 3 2 2 2 2
                                           % 3 3 2 2 2 2
% {W=S1+S2}, {S2=<1}, {S1=S3},             % 1 1 2 2 2 2
% {S2>=S1+S3}, {S1+S3>=1}.                 % 1 1 2 2 2 2

S1 = 1, S2 = 1/2, S3 = 1/2, W = 3/2 ? ;    % 1 1 1 1 3 3
                                           % 1 1 1 1 3 3
% {W=S1+S2}, {S2=S3}, {S2+S3=<1},          % 1 1 1 1 2 2
% {S2+S3>=S1}, {S1>=1}.                    % 1 1 1 1 2 2

S1 = 1, S2 = 1, S3 = 1, W = 3 ? ; no

% {W=S1+S2+S3}, {S3=<1}, {S3>=S2},         % 1 1 2 2 3 3
% {S2>=S1}, {S1>=1}.                       % 1 1 2 2 3 3
```


Tökéletes téglalapok: választási pontok

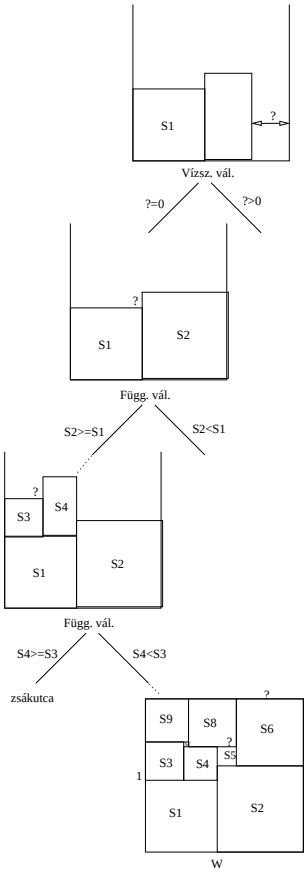
Függőleges



Vízszintes



Tökéletes téglalapok: a keresési tér szerkezete



A korlát logikai programozás elmélete

Egy CLP rendszer

- $\langle \mathcal{D}, \mathcal{F}, \mathcal{R}, \mathcal{S} \rangle$
- $\mathcal{D}$: egy tartomány (domain), pl. egészek (N), valósak (R), racionálisak(Q), Boole értékek (B), listák, füzérek (stringek) (+ a Prolog-fastruktúrák (Herbrand — H) tartománya)
- $\mathcal{F}$: $\mathcal{D}$-ben definiált függvényjeleknek egy halmaza, pl. $+$, $-$, $*$, $\vee$, $\wedge$
- $\mathcal{R}$: $\mathcal{D}$-ben definiált relációjeleknek (korlátoknak) egy halmaza pl. $=$, $\neq$, $<$, $\in$
- $\mathcal{S}$: egy korlát-megoldó algoritmus $\langle \mathcal{D}, \mathcal{F}, \mathcal{R} \rangle$-re, azaz a $\mathcal{D}$ tartományban az $\mathcal{F} \cup \mathcal{R}$ halmazbeli jelekből felépített korlátokra

CLP szintaxis és deklaratív szemantika

program

- klózok halmaza.

klóz

- szintaxis: `P :- G1, ..., Gn`, ahol mindegyik $\mathtt{G}_i$ vagy eljáráshívás, vagy korlát.
- deklaratív olvasat: `P` igaz, ha $\mathtt{G}_1, \ldots, \mathtt{G}_n$ mind igaz.

kérdés

- szintaxis: `?- G1, ..., Gn`
- válasz egy `Q` kérdésre: korlátoknak egy olyan konjunkciója, amelyből a kérdés következik.

CLP procedurális szemantika

Végrehajtási állapot

- $\langle \mathtt{G}, s \rangle$
- `G` — cél/korlát sorozat
- s — korlát-tár: az eddig felhalmozott egyszerű korlátok konjunkciója (kezdetben üres)

Szükséges megkülönböztetés

- egyszerű korlát (`C`): amit a korlát-tár közvetlenül befogad ($\mathcal{F} \cup \mathcal{R}$-től függ)
- összetett korlát (`C`): a tár nem tudja befogadni, de hathat a tárra

Klózok procedurális olvasata

- `P :- G1, ..., Gn` jelentése: `P` megoldásához megoldandó $\mathtt{G}_1, \ldots, \mathtt{G}_n$.

Végrehajtási invariánsok

- s konzisztens
- $\mathtt{G} \wedge s \rightarrow \mathtt{Q}$ (`Q` a kezdő kérdés)

Végrehajtás vége

- $\langle \mathtt{G}_e, s_e \rangle$, ahol $\mathtt{G}_e$-re nem alkalmazható egyetlen következtetési lépés sem.

A végrehajtás eredménye

- Az s_e korlát-tár, vagy annak a kérdésben szereplő változókra való „vetítése” (a többi változó egzisztenciális kvantálásával).
- A $\mathtt{G}_e$ fennmaradó (összetett) korlátok.

A CLP következtetés folyamata

Következtetési lépések

- rezolúció:
  $\langle \mathtt{P} \,\&\, \mathtt{G}, s \rangle \Rightarrow \langle \mathtt{G}_1 \,\&\, \ldots \,\&\, \mathtt{G}_n \,\&\, \mathtt{G}, \mathtt{P} = \mathtt{P}' \wedge s \rangle$,
  feltéve, hogy a programban van egy `P' :- G1, ..., Gn` klóz
- korlát-megoldás:
  $\langle \mathtt{C} \,\&\, \mathtt{G}, s \rangle \Rightarrow \langle \mathtt{G}, s \wedge \mathtt{C} \rangle$
- korlát-erősítés:
  $\langle \mathtt{C} \,\&\, \mathtt{G}, s \rangle \Rightarrow \langle \mathtt{C}' \,\&\, \mathtt{G}, s \wedge \mathtt{C} \rangle$
  ha s-ből következik, hogy `C` ekvivalens ($\mathtt{C}' \wedge \mathtt{C}$)-vel. ($\mathtt{C}' = \mathtt{C}$ is lehet.)

Ha a tár inkonzisztensé válna, visszalépés történik.

Példa erősítésre

- $\langle \mathtt{X > Y*Y} \,\&\, \ldots, \mathtt{Y > 3} \rangle \Rightarrow \langle \mathtt{X > Y*Y} \,\&\, \ldots, \mathtt{Y > 3} \wedge \mathtt{X > 9} \rangle$
  hiszen $\mathtt{X > Y*Y} \wedge \mathtt{Y > 3} \Rightarrow \mathtt{X > 9}$
- clp(R)-ben nincs ilyen, de clp(FD)-ben van!

Követelmények a korlát megoldó algoritmussal szemben

- teljesség (egyszerű korlátok konjunkciójáról mindig döntse el, hogy konzisztens-e),
- inkrementalitás (az s tár konzisztenciáját ne bizonyítsa újra),
- a visszalépés támogatása,
- hatékonyság.

A `clpb` könyvtár

- **Tartomány:** logikai értékek (`1` és `0`, igaz és hamis)
- **Függvények** (egyben korlát-relációk):

`~ P`	`P` hamis (*negáció*).
`P * Q`	`P` és `Q` mindegyike igaz (*konjunkció*).
`P + Q`	`P` és `Q` legalább egyike igaz (*diszjunkció*).
`P # Q`	`P` és `Q` pontosan egyike igaz (*kizáró vagy*).
`X ^ P`	Létezik olyan `X`, hogy `P` igaz (azaz `P[X/0]+P[X/1]` igaz).
`P =\= Q`	Ugyanaz mint `P # Q`.
`P =:= Q`	Ugyanaz mint `~(P # Q)`.
`P =< Q`	Ugyanaz mint `~P + Q`.
`P >= Q`	Ugyanaz mint `P + ~Q`.
`P < Q`	Ugyanaz mint `~P * Q`.
`P > Q`	Ugyanaz mint `P * ~Q`.
`card(Is, Es)`	Az `Es` listában szereplő igaz értékű kifejezések száma eleme az `Is` által jelölt halmaznak (`Is` egészek és `Tol-Ig` szakaszok listája).

- **Primitív korlátok** (korlát tár elemei): tetszőleges korlát (Boole-egyesítők formájában).
- **Korlát-megoldó algoritmus:** Boole-egyesítés.

A `library(clpb)` könyvtár eljárásai

- `sat(`*Kifejezés*`)`, ahol *Kifejezés* változókból, a `0`, `1` konstansokból és atomokból (ún. szimbolikus konstansok) a fenti müveletekkel felépített logikai kifejezés. Hozzáveszi *Kifejezés*t a korlát-tárhoz.
- `taut(`*Kif*`,` *Ért*`)`. Megvizsgálja, hogy *Kif* **levezethető-e** a tárból, ekkor *Ért*=`1`; vagy negáltja levezethető-e, ekkor *Ért*=`0`. Egyébként meghiúsul.
- `labeling(`*Változók*`)`. Behelyettesíti a *Változók*at `0`, `1` értekekre (úgy, hogy a tár teljesüljön). Visszalépéskor felsorolja az összes lehetséges értéket.

Egyszerű példák

```
| ?- sat(X + Y).                          sat(X=\=_A*Y#Y) ?

| ?- sat(x + Y).                          sat(Y=\=_A*x#x) ?

| ?- taut(_A ^ (X=\=_A*Y#Y) =:= X+Y, T).
                                          T = 1 ?

| ?- sat(A # B =:= 0).                    B = A ?

| ?- sat(A # B =:= C), A = B.             B = A, C = 0 ?

| ?- taut(A =< C, T).                     no

| ?- sat(A =< B), sat(B =< C), taut(A =< C, T).
                                          T = 1,
                                          sat(A=:=_A*_B*C),
                                          sat(B=:=_B*C) ?
```

Megjegyzések

- A tár megjelenítése: `sat(V =:= Kif)` ill. `sat(V =\= Kif)` ahol `Kif` egy „polinom”, azaz konjunkciókból kizáró vagy (#) müvelettel képzett kifejezés.
- Az atommal jelölt szimbolikus konstansok nem behelyettesíthetőek, (legkívül) univerzálisan kvantifikált változóknak tekinthetők.

```
| ?- sat(~x+ ~y=:= ~(x*y)).    % ∀xy(¬x ∨ ¬y = ¬(x ∧ y))
                                   yes
| ?- sat(~X+ ~Y=:= ~(X*Y)).    % ∃?XY(¬X ∨ ¬Y = ¬(X ∧ Y))
                                   true ? ; no
| ?- sat(x=<y).                % ∀xy(x → y)
                                   no
| ?- sat(X=<y).                % ∀y∃?X(X → y)
                                   sat(X=:=_A*y) ? ; no
```


Példa: 1-bites összeadó

```
| ?- [user].
| adder(X, Y, Sum, Cin, Cout) :-
     sat(Sum =:= card([1,3],[X,Y,Cin])),
     sat(Cout =:= card([2-3],[X,Y,Cin])).
| {user consulted, 40 msec 576 bytes}

yes
| ?- adder(x, y, Sum, cin, Cout).

sat(Sum=:=cin#x#y),
sat(Cout=:=x*cin#x*y#y*cin) ?

yes
| ?- adder(x, y, Sum, 0, Cout).

sat(Sum=:=x#y),
sat(Cout=:=x*y) ?

yes
| ?- adder(X, Y, 0, Cin, 1), labeling([X,Y,Cin]).

Cin = 0, X = 1, Y = 1 ? ;

Cin = 1, X = 0, Y = 1 ? ;

Cin = 1, X = 1, Y = 0 ? ;

no
```


Boole-egyesítés

A feladat:

- Adott `g` és `h` logikai kifejezések.
- Keressük a `g = h` egyenletet megoldó legáltalánosabb egyesítőt (mgu).
- Példa: mgu(`X+Y`, `1`) lehet `X = W * Y # Y # 1` (új változó, pl. `W`, bejöhet).
- Egyszerűsítés: A `g = h` egyenlet helyettesíthető az `f = 0` egyenlettel, ahol `f = g # h`.
- Az egyesítés során minden lépésben egy f `= 0` formulabeli változót szeretnénk kifejezni.

Az `X` változó kifejezése

- Legyen $f_X(1)$ az f-ből az `X=1`, $f_X(0)$ az `X=0` behelyettesítéssel kapott kifejezés.
- f `= 0` kielégíthetőségének szükséges feltétele $f_X(1)$ `*` $f_X(0)$ `= 0` kielégíthetősége.
- Fejezzük ki `X`-et $f_X(0)$-val és $f_X(1)$-gyel úgy, hogy f `= 0` legyen!

$f_X(0)$	$f_X(1)$	`X`
0	0	bármi (`W`)
0	1	0
1	0	1
1	1	érdektelen

  Keressük `X`-et `X = A*~W # B*W` alakban!
- Határozzuk meg `A`-t és `B`-t $f_X(0)$ és $f_X(1)$ függvényeként!

$f_X(0)$	$f_X(1)$	`X`	`A`	`B`
0	0	`W`	0	1
0	1	0	0	0
1	0	1	1	1

  Az `A` = $f_X(0)$ és `B` = ~$f_X(1)$ megfeleltetés tűnik a legegyszerűbbnek.

Boole-egyesítés (folyt.)

Az egyesítési algoritmus az f `= 0` egyenlőségre

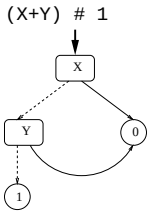
- Ha f-ben nincs változó, akkor azonosnak kell lennie `0`-val (különben nem egyesíthető).
- Helyettesítsünk: `X = ~W*`$f_{\texttt{X}}(0)$ `# W*~`$f_{\texttt{X}}(1)$ (Boole-egyesítő)
- Folytassuk az egyesítést az $f_{\texttt{X}}(1)$ `*` $f_{\texttt{X}}(0)$ `= 0` egyenlőségre.

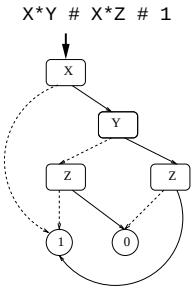
Példák

- mgu(X+Y, 0) $\longrightarrow$ `X = 0, Y = 0`;
- mgu(X+Y, 1) = mgu(`~(X+Y)`, 0) $\longrightarrow$ `X = W * Y # Y # 1`;
- mgu(X*Y, `~(X*Z)`) = mgu(`(X*Y)#(X*Z)#1`, 0) $\longrightarrow$ `X = 1, Y = ~Z`.

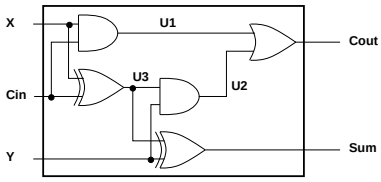
Belső ábrázolás: BDD (Boolean/Binary Decision Diagrams)

(Szaggatott vonal: 0 érték, folytonos vonal: 1 érték)



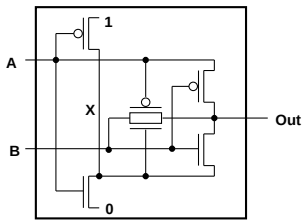


Példa: Hibakeresés áramkörben




```
fault([F1,F2,F3,F4,F5], [X,Y,Cin], [Sum,Cout]) :-
        sat(
                    card([0-1],[F1,F2,F3,F4,F5]) *
                    (F1 + (U1 =:= X * Cin)) *
                    (F2 + (U2 =:= Y * U3)) *
                    (F3 + (Cout =:= U1 + U2)) *
                    (F4 + (U3 =:= X # Cin)) *
                    (F5 + (Sum =:= Y # U3))
                ).

| ?- fault(L, [1,1,0], [1,0]).
                               L = [0,0,0,1,0] ? ;  no

| ?- fault(L, [1,0,1], [0,0]).
                               L = [_A,0,_B,0,0],
                               sat(_A=\=_B) ? ; no

| ?- fault(L, [1,0,1], [0,0]), labeling(L).
                               L = [1,0,0,0,0] ? ;
                               L = [0,0,1,0,0] ? ; no

| ?- fault([0,0,0,0,0], [x,y,cin], [Sum,Cout]).
                               sat(Cout=:=x*cin#x*y#y*cin),
                               sat(Sum=:=cin#x#y) ? ; no
```


Példa: Tranzisztoros áramkör verifikálása




```
n(D, G, S) :-    % Gate => Drain = Source
        sat( G*D =:= G*S).

p(D, G, S) :-    % ~ Gate => Drain = Source
        sat( ~G*D =:= ~G*S).

xor(A, B, Out) :-
        p(1, A, X),
        n(0, A, X),
        p(B, A, Out),
        n(B, X, Out),
        p(A, B, Out),
        n(X, B, Out).

| ?- n(D, 1, S).                S = D ?

| ?- n(D, 0, S).                true ?

| ?- p(D, 0, S).                S = D ?

| ?- p(D, 1, S).                true ?

| ?- xor(a, b, X).              sat(X=:=a#b) ?
```


Minesweeper `clpb`-ben

```
:- use_module([library(clpb),library(lists)]).

mine(Rows, Cols, Mines, Bd) :-
        length(Bd, Rows), all_length(Bd, Cols),
        append_lists(Bd, All),
        sat(card([Mines], All)), play_mine(Bd, []).

all_length([], _).
all_length([L|Ls], Len) :-
        length(L, Len), all_length(Ls, Len).

append_lists([], []).
append_lists([L|Ls], Es) :-
        append_lists(Ls, Es0), append(L, Es0, Es).

play_mine(Bd, Asked) :-
        select_field(Bd, Asked, R, C, E), !,
        format('Row ~w, col ~w (m for mine)? ', [R,C]),
        read(Ans), process_ans(Ans, E, R, C, Bd),
        play_mine(Bd, [R-C|Asked]).
play_mine(_Bd, _Asked).

select_field(Bd, Asked, R, C, E) :-
        nth(R, Bd, L), nth(C, L, E),
        non_member(R-C, Asked), taut(E, 0), !.
select_field(Bd, _Asked, R, C, E) :-
        nth(R, Bd, L), nth(C, L, E),
        non_member(R-C, Asked), \+ taut(E,1), !.

process_ans(m, 1, _, _, _) :-
        format('Mine!~n', []), !, fail.
process_ans(Ans, 0, R, C, Bd) :-
        integer(Ans), neighbs(n(R, C, Bd), Ns),
        sat(card([Ans], Ns)).

neighbs(RCB, N7) :-
        neighbour(-1,-1, RCB, [], N0),
        neighbour(-1, 0, RCB, N0, N1),
        neighbour(-1, 1, RCB, N1, N2),
        neighbour( 0,-1, RCB, N2, N3),
        neighbour( 0, 1, RCB, N3, N4),
        neighbour( 1,-1, RCB, N4, N5),
        neighbour( 1, 0, RCB, N5, N6),
        neighbour( 1, 1, RCB, N6, N7).

neighbour(ROf, COf, n(R0, C0, Bd), Nbs, [E|Nbs]) :-
        R is R0+ROf, C is C0+COf,
        nth(R, Bd, Row), nth(C, Row, E), !.
neighbour(_, _, _, Nbs, Nbs).
```