Nagyhatékonyságú logikai programozás

Jegyzetek a BME informatikus hallgatói számára

Szeredi Péter, Zombori Zsolt

Számítástudományi
és Információelméleti Tanszék

{szeredi,zombori}@cs.bme.hu

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

Budapest 2011. 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/`)

Információk a Mercury nyelvről

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

A CLP alapgondolata

A CLP($\mathcal{X}$) séma

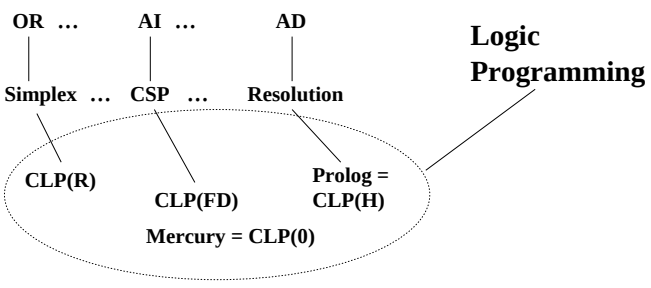
Prolog + egy valamilyen $\mathcal{X}$ adattartományra és azon értelmezett korlátokra (relációkra) vonatkozó „erős” következtetési mechanizmus.

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
(Motiváció: a CLP alapelvek és egyben a haladó Prolog lehetőségek bemutatása.)

- 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

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

Listák biztonságos összefűzése blokk-deklaráció segítségével

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


| ?- app(L1, L2, L3).
user:app(L1,L2,L3) ? ;
no
| ?- app(L1, L2, L3), L3 = [a|L4].
L1 = [], L2 = [a|L4], L3 = [a|L4] ? ;
L1 = [a|_A], L3 = [a|L4], user:app(_A,L2,L4) ? ;
no
```


Listák biztonságos összefűzése, nyomkövetés

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


| ?- trace, app(L1, L2, L3), L3 = [a|L4], L4 = [].
% The debugger will first creep -- showing everything (trace)
        -      - Block: app(_1012,_532,_1018)
        1      1 Call: _1018=[a|_622] ?
        -      - Unblock: app(_1012,_532,[a|_622])
        2      2 Call: app(_1012,_532,[a|_622]) ?
?       2      2 Exit: app([],[a|_622],[a|_622]) ?
?       1      1 Exit: [a|_622]=[a|_622] ?
        3      1 Call: _622=[] ?
        3      1 Exit: []=[] ?
L1 = [], L2 = [a], L3 = [a], L4 = [] ? ;
        1      1 Redo: [a|_622]=[a|_622] ?
        2      2 Redo: app([],[a|_622],[a|_622]) ?
        -      - Block: app(_2098,_532,_2104)
        2      2 Exit: app([a|_2098],_532,[a|_2104]) ? &
Blocked goals:
1 (_2098): user:app(_2098,_532,_2104)
2 (_2104): user:app(_2098,_532,_2104)
        2      2 Exit: app([a|_2098],_532,[a|_2104]) ?
        1      1 Exit: [a|_2104]=[a|_2104] ?
        4      1 Call: _2104=[] ?
        -      - Unblock: app(_2098,_532,[])
        5      2 Call: app(_2098,_532,[]) ?
?       5      2 Exit: app([],[],[]) ?
?       4      1 Exit: []=[] ?
L1 = [a], L2 = [], L3 = [a], L4 = [] ? ;
        4      1 Redo: []=[] ?
        5      2 Redo: app([],[],[]) ?
        5      2 Fail: app(_2098,_532,[]) ?
        4      1 Fail: _2104=[] ?
no
```


Példa korutinszervezésre: többirányú összeadás

```
% plusz(X, Y, Z): X+Y=Z,  ahol X, Y és Z természetes számok.
% Bármelyik argumentum lehet behelyettesítetlen.
plusz(X, Y, Z) :-
        app(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) :-
        Len1 is Len0+1, len(L, Len1, Len).
len([], Len, 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 (`process` csak akkor hívódik meg, ha `T` tömör, és vagy `X` nem változó, vagy `X` és `Y` egyesíthetősége eldönthető):

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

A `dif` eljárás a `when` segítségével definiálható:

```
dif(X, Y) :- when(?=(X,Y), X\==Y).
```

Késleltetett hívások lekérdezése

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

`call_residue_vars(Hivas, Valtozok)`
`Hivas`-t végrehajtja, és a `Valtozok` listában visszaadja mindazokat az új (a `Hivas` alatt létrejött) változókat, amelyekre vonatkoznak felfüggesztett hívások. Pl.

```
| ?- call_residue_vars((dif(X,f(Y)), X=f(Z)), Vars).

     X = f(Z),
     Vars = [Z,Y],
     prolog:dif(f(Z),f(Y)) ?
```

Többirányú összeadás when segítségével

```
:- use_module(library(between)).

% app(L1, L2, L3): L1 és L2 összefűzöttje L3.
% ahol L1, L2 és L3 1-es számokból álló listák.
app([], L, L).
app([1|L1], L2, [1|L3]) :-
    when((nonvar(L1);nonvar(L3)),
         app(L1, L2, L3)).

len(L, Len) :-
        when(ground(L), length(L, Len)),
        when(nonvar(Len), findall(1, between(1, Len, _), L)).

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

| ?- 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
```


CLP(MiniNat) megvalósítása

Számábrázolás

- A korábbi `plusz/3` eljárásban egy N elemű listával ábrázoltuk az N számot (a listaelemek érdektelenek, behelyettesítetlen változók vagy 1-esek)
- Példa: a 2 szám ábrázolása: `[_,_]` $\equiv$ `.(_,.(_,[]))`.
- Hagyjuk el a felesleges listaelemeket, 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 = s(_), +(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) :-
        szor(X, Y, Z1),
        +(Z1, Y, Z).
```


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), % Kif1 értékét E-ben
                                % előállító cél 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).

Példa: mátrixok kiíratása

```
portray(Matrix) :-
     Matrix = [[_|_]|_],
     (   member(Row, Matrix), nl, print(Row), fail
     ;   true
     ).

| ?- X = [[1,2,3],[4,5,6]].
X =
[1,2,3]
[4,5,6] ?
```


Példa testreszabott kiíratásra: Peano számok

```
% 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 =.. [Pred,A,B,C],
        predikatum_operator(Pred, Op),
        Fun =.. [Op,A,B],
        print({Fun=C}).

predikatum_operator(plusz, +).
predikatum_operator(+, +).
predikatum_operator(*, *).
```


Prolog háttér: programok előfeldolgozása

Kampó (Hook, callback) eljárások a fordítási idejű átalakításhoz:

- `user:term_expansion(+Kif, ..., -Klózok, ...)`: (közelítő leírás:) Minden betöltő eljárás (`consult, compile` stb.) által beolvasott kifejezésre a rendszer meghívja. A kimenő paraméterben várja a transzformált alakot (lehet lista is). Meghiúsulás esetén változtatás nélkül veszi fel a kifejezést klózként.
- `M:goal_expansion(+Cél, +Layout, +Modul, -ÚjCél, -ÚjLayout)`: Minden a beolvasott programban (vagy feltett kérdésben) előforduló részcélra meghívja a rendszer. A kimenő paraméterekben várja a transzformált alakot (lehet konjunkció). Meghiúsulás esetén változtatás nélkül hagyja a célt. (Ha a forrásszintű nyomkövetés nem fontos, `ÚjLayout` lehet `[]`.)

CLP(MiniNat) továbbfejlesztése `goal_expansion` használatával

- A funkcionális alak átalakítása a betöltés alatt is elvégezhető (kompilálás):

```
goal_expansion({Korlat}, _LO, _Module, Cel, /*ÚjLO*/ []) :-
        korlat_cel(Korlat, Cel).
```

- Célszerű a generált célsorozatból a `true` hívásokat kihagyni.

```
% osszetett(C1, C2, C): C a C1 és C2 célok konjunkciója.
osszetett(true, Cel0, Cel) :-    !, Cel = Cel0.
osszetett(Cel0, true, Cel) :-    !, Cel = Cel0.
osszetett(Cel1, Cel2, (Cel1,Cel2)).
```

- A fenti eljárást használjuk a konjunkciók helyett, pl:

```
korlat_cel((K1,K2), C12) :-
         korlat_cel(K1, C1), korlat_cel(K2, C2),
         osszetett(C1, C2, C12).
```

Megjegyzés: a faktoriális példában ez a kompilálás 6-7% gyorsulást jelent

Előfeldolgozás a faktoriális példa esetén

- A faktoriális példa betöltött alakja :

```
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, 6).         -->  no
| ?- {F=6}, fact(N, F).  -->  F = 6, N = 3 ? ; no
```


CLP(MiniNat) használata — példák

(Kompilálás nélkül)

```
:- 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, 12).
no

| ?- fact(F, 15).
! Resource error: insufficient memory

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


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: kiszámolja `10!`-t, és csak ezután egyesíti `1`-gyel. Azonban a `10!` kiszámolásához (Peano számként) sok idő és memória kell `:-(`.
- A probléma javítása: a szorzat-feltételt tegyük a rekurzív `fact/2` hívás elé. Egy további gyorsítási lehetőség a *redundáns* korlátok alkalmazása.

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

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

- Azonban az alábbi cél futása még így is kivárhatatlan . . .

```
| ?- fact(N, 5040).   -------->   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` értékekre. 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
```


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 elimináció, 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
```

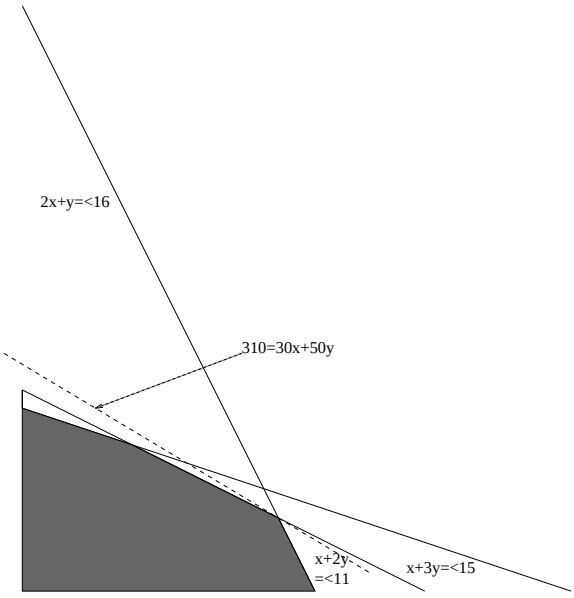
`bb_inf(Egészek, Kif, Inf)` — kiszámolja `Kif` infimumát, azzal a további feltétellel, hogy az Egészek listában levő minden változó egész (ún. „Mixed Integer Optimisation Problem”).

```
| ?- {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.

További eljárások (lásd kézikönyv): `bb_inf/5`, `dump/3`, `projecting_assert/1`,

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:

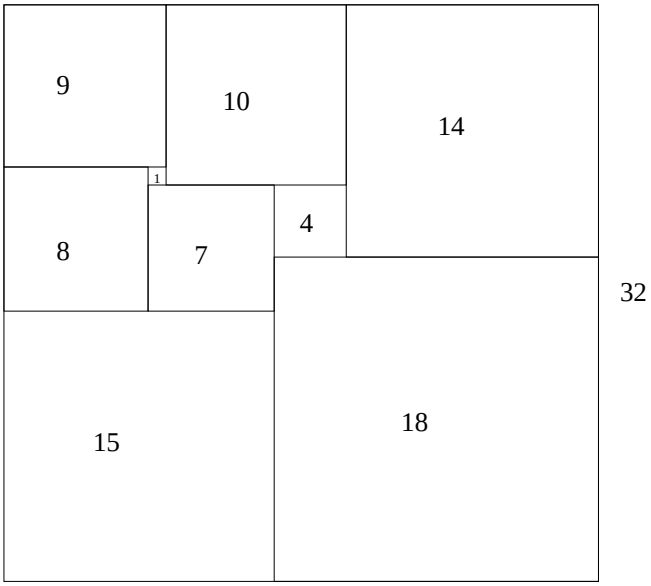
```
| ?- {X=0.5}, X=0.5.
no
| ?- {X=0.5}, X=1/2.
no
| ?- {X=0.5}, X=rat(2,4).
no
| ?- {X=0.5}, X=rat(1,2).
X = 1/2 ?
| ?- {X=5}, X=5.
no
| ?- {X=5}, X=rat(5,1).
X = 5 ?
```


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] ?
```

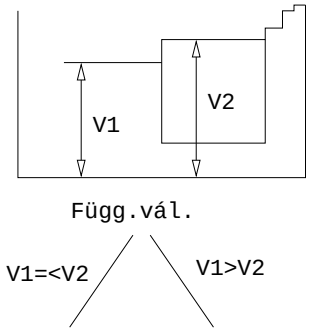
Az `outof` hívás kihagyásával végzett futtatás

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

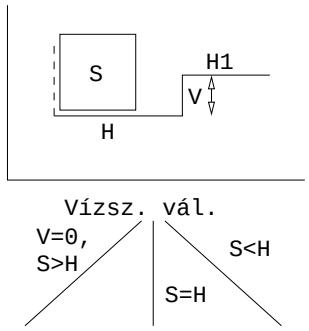
```
| ?- filled_rectangle(W, [S1,S2,S3], [eqsq]).

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

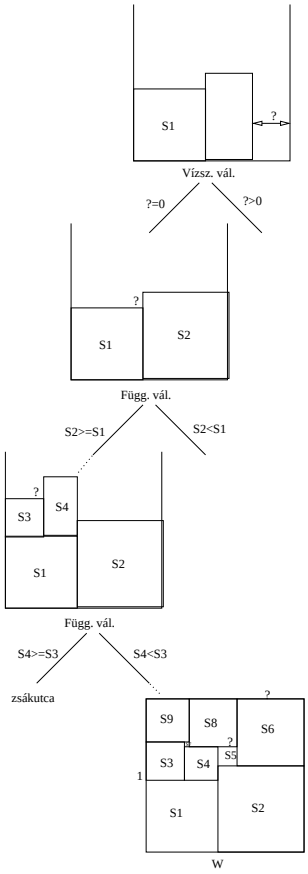
Függőleges



Vízszintes



Tökéletes téglalapok: a keresési tér szerkezete



A 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 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 :-` $\mathtt{G}_1, \ldots, \mathtt{G}_n$, 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: `?-` $\mathtt{G}_1, \ldots, \mathtt{G}_n$
- válasz egy `Q` kérdésre: korlátoknak egy olyan konjunkciója, amelyből a kérdés következik.

CLP procedurális szemantika

Végrehajtási állapot

- $\langle$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 :- $G_1, \ldots, G_n$ jelentése: P megoldásához megoldandó $G_1, \ldots, G_n$.

Végrehajtási invariánsok

- s konzisztens
- G $\wedge$ $s \rightarrow$ Q (Q a kezdő kérdés)

Végrehajtás vége

- $\langle G_e, s_e\rangle$, ahol 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 G_e fennmaradó (összetett) korlátok.

A CLP következtetés folyamata

Következtetési lépések

- rezolúció:
  $\langle$P & G, $s\rangle \Rightarrow \langle G_1$ & … & G_n & G, P = P′ $\wedge$ $s\rangle$,
  feltéve, hogy a programban van egy P′ :- $G_1, \ldots, G_n$ klóz
- korlát-megoldás:
  $\langle$C & G, $s\rangle \Rightarrow \langle$G, $s \wedge$ C$\rangle$
- korlát-erősítés:
  $\langle$C & G, $s\rangle \Rightarrow \langle$C′ & G, $s \wedge$ C$\rangle$
  ha s-ből következik, hogy C ekvivalens (C′ $\wedge$ C)-vel. (C′ = C is lehet.)

Ha a tár inkonzisztensé válna, visszalépés történik.

Példa erősítésre

- $\langle$X > Y*Y & …, Y > 3$\rangle \Rightarrow \langle$X > Y*Y & …, Y > 3 $\wedge$ X > 9$\rangle$
  hiszen X > Y*Y $\wedge$ Y > 3 $\Rightarrow$ 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).

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

$f_{\mathtt{X}}(0)$	$f_{\mathtt{X}}(1)$	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 $f_{\mathtt{X}}(0)$ és $f_{\mathtt{X}}(1)$ függvényeként!

$f_{\mathtt{X}}(0)$	$f_{\mathtt{X}}(1)$	X	A	B
0	0	`W`	0	1
0	1	0	0	0
1	0	1	1	1

  Az `A =` $f_{\mathtt{X}}(0)$ és `B =` `~`$f_{\mathtt{X}}(1)$ 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*`$f_{\mathtt{X}}(0)$ `# W*~`$f_{\mathtt{X}}(1)$ (Boole-egyesítő)
- Folytassuk az egyesítést az $f_{\mathtt{X}}(1)$ `*` $f_{\mathtt{X}}(0)$ `= 0` egyenlőségre.

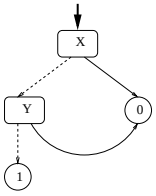
Példák

- 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`.

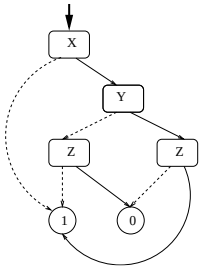
Belső ábrázolás: BDD (Boolean/Binary Decision Diagrams)

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

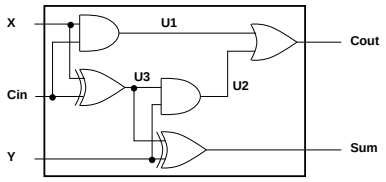
`(X+Y) # 1`



`X*Y # X*Z # 1`

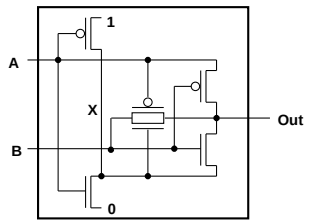


Példa: Hibakeresés áramkörben




```
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).
```


A SICStus `clpfd` könyvtár

Tartomány

Egészek (negatívak is) véges (esetleg végtelen) halmaza

Korlátok

- aritmetikai
- logikai
- halmaz (halmazba tartozás)
- kombinatorikai
- tükrözött
- felhasználó által definiált

Egyszerű korlátok

csak a halmaz-korlátok: `X ∈ Halmaz`

Korlát-megoldó algoritmus

- egyszerű korlátok kezelése triviális;
- a lényeg az összetett korlátok **erősítő** tevékenysége, ez a Mesterséges Intelligencia CSP (Constraint Satisfaction Problems) ágának módszerein alapul.

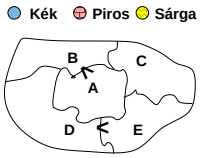
Miről lesz szó?

- CSP, mint háttér
- Alapvető (aritmetikai és halmaz-) korlátok
- Tükrözött és logikai korlátok
- Címkéző eljárások
- Kombinatorikai korlátok
- Felhasználó által definiált korlátok: indexikálisok és globális korlátok
- Az FDBG nyomkövető csomag
- Esettanulmányok: négyzetdarabolás, torpedó-, ill, dominó-feladvány

Háttér: CSP (Constraint Satisfaction Problems)

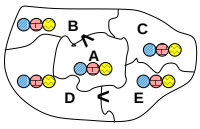
Példafeladat

Az alábbi térkép kiszínezése kék, piros és sárga színekkel úgy, hogy a szomszédos országok különböző színűek legyenek, és ha két ország határán a < jel van, akkor a két szín ábécé-rendben a megadott módon kövesse egymást.

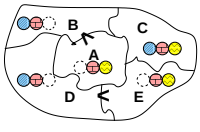


Egy lehetséges megoldási folyamat (*zárójelben a CSP elnevezések*)

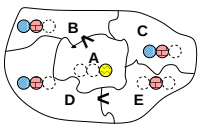
1. Minden mezőben elhelyezzük a három lehetséges színt (*változók és tartományaik felvétele*).



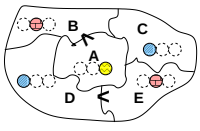
2. Az „A” mező nem lehet kék, mert annál „B” nem lehetne kisebb. A „B” nem lehet sárga, mert annál „A” nem lehetne nagyobb. Az „E” és „D” mezők hasonlóan szűkíthetők (*szűkítés, él-konzisztencia biztosítása*).



3. Ha az „A” mező piros lenne, akkor mind „B”, mind „D” kék lenne, ami ellentmondás (*globális korlát, ill. borotválási technika*). Tehát „A” sárga. Emiatt a vele szomszédos „C” és „E” nem lehet sárga (*él-konszitens szűkítés*).



4. „C” és „D” nem lehet piros, tehát kék, így „B” csak piros lehet (*él-konszitens szűkítés*). Tehát az egyetlen megoldás:
A = sárga, B = piros, C = kék, D = kék, E = piros.



A CSP problémakör rövid áttekintése

A CSP fogalma

- CSP = (X, D, C)
  - $X = \langle x_1, \ldots, x_n \rangle$ — változók
  - $D = \langle D_1, \ldots, D_n \rangle$ — tartományok, azaz nem üres halmazok
  - x_i változó a D_i véges halmazból (x_i tartománya) vehet fel értéket
  - C a problémában szereplő korlátok (atomi relációk) halmaza, argumentumaik X változói (például $C \ni c = r(x_1, x_3)$, $r \subseteq D_1 \times D_3$)
- A CSP feladat megoldása: minden x_i változóhoz egy $v_i \in D_i$ értéket kell rendelni úgy, hogy minden $c \in C$ korlátot egyidejűleg kielégítsünk.
- **Definíció:** egy c korlát egy x_i változójának d_i értéke *felesleges*, ha nincs a c többi változójának olyan értékrendszere, amely d_i-vel együtt kielégíti c-t.
- **Állítás:** *felesleges érték elhagyásával (szűkítés) ekvivalens CSP-t kapunk.*
- **Definíció:** egy korlát *él-konzisztens* (arc consistent), ha egyik változójának tartományában sincs felesleges érték. A CSP *él-konzisztens*, ha minden korlátja él-konzisztens. Az él-konzisztencia szűkítéssel biztosítható.
- Ha minden reláció bináris, a CSP probléma gráffal ábrázolható (változó ⇒ csomópont, reláció ⇒ él). Az *él*-konzisztencia elnevezés ebből fakad.

A CSP megoldás folyamata

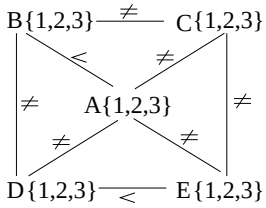
- felvesszük a változók tartományait;
- felvesszük a korlátokat mint démonokat, amelyek szűkítéssel él-konzisztenciát biztosítanak;
- többértelműség esetén címkézést (labeling) végzünk:
  - kiválasztunk egy változót (pl. a legkisebb tartományút),
  - a tartományt két vagy több részre osztjuk (választási pont),
  - az egyes választásokat visszalépéses kereséssel bejárjuk (egy tartomány üresre szűkülése váltja ki a visszalépést).

A térképszínezés mint CSP feladat

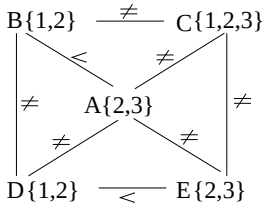
Modellezés (leképezés CSP-re)

- változók meghatározása: országonként egy változó, amely az ország színét jelenti;
- változóértékek kódolása: kék → `1`, piros → `2`, sárga → `3` (sok CSP megvalósítás kiköti, hogy a tartományok elemei pl. nem-negatív egészek);
- korlátok meghatározása:
  - az előírt `<` relációk teljesülnek,
  - a többi szomszédos ország-pár különböző színű.

A kiinduló korlát-gráf



A korlát-gráf él-konzisztens szűkítése



CLP(FD) = a CSP beágyazása a CLP($\mathcal{X}$) sémába

A CSP → CLP(FD) megfeleltetés

- CSP változó → CLP változó
- CSP: x tartománya T → CLP: „`X in T`” egyszerű korlát.
- CSP korlát → CLP korlát, *általában összetett!*

A CLP(FD) korlát-tár

- Tartalma: *X* `in` *Tartomány* alakú egyszerű korlátok.
- Tekinthető úgy mint egy hozzárendelés a változók és tartományaik (lehetséges értékek) között.
- Egyszerű korlát hozzávétele a tárhoz: egy már bennlévő változó tartományának szűkítése vagy egy új változó-hozzárendelés felvétele.

Összetett CLP(FD) korlátok

- A korlátok többsége démon lesz, hatását a *korlát-erősítés*en keresztül fejti ki ($\langle C, s \rangle \longrightarrow \langle C', s \wedge c \rangle$ ahol $s \models C \equiv C' \wedge c$).
- Az erősítés egy egyszerű korlát hozzávételét, azaz a CLP(FD) esetén a tár szűkítését jelenti.
- A démonok ciklikusan működnek: szűkítenek, elalszanak, aktiválódnak, szűkítenek, ...
- A démonokat a korlátbeli változók tartományának változása aktiválja.
- Különböző korlátok különböző mértékű szűkítést alkalmazhatnak (a maximális szűkítés túl drága lehet).

A `clpfd` könyvtár — alapvető-korlátok

Alapvető aritmetikai korlátok

- Függvények

  `+ - * / mod min max` (kétargumentumúak),
  `abs` (egyargumentumú).
- Korlát-relációk: `#<, #>, #=<, #>=, #= #\=` (mind `xfx 700` operátorok)

Halmazkorlátok

- *`X in KTartomány`*, jelentése: $X \in H$, ahol H a *`KTartomány`* (konstans tartomány) által leírt halmaz (Az `in` atom egy `xfx 700` operátor);
- `domain([`*`X,Y,...`*`],`*`Min,Max`*`)`: $X \in$ [*`Min,Max`*], $Y \in$ [*`Min,Max`*], ...

Itt *`Min`* lehet *`Szám`* vagy `inf` ($-\infty$), *`Max`* pedig *`Szám`* vagy `sup` ($+\infty$);
(Megjegyzés: a végtelen tartományok főleg kényelmi célokat szolgálnak: nem kell kiszámolnunk az alsó/felső korlátokat, ha azok kikövetkeztethetők.)

Egy *`KTartomány`* a következők egyike lehet:

- felsorolás: {*`Szám`*, ...},
- intervallum: (*`Min..Max`*), (`xfx 550` operátor),
- metszet: *`KTartomány`* `/\` *`KTartomány`* (`yfx 500`, beépített op.),
- únió: *`KTartomány`* `\/` *`KTartomány`*, (`yfx 500`, beépített op.),
- komplemens: `\` *`KTartomány`*, (`fy 500` operátor).

Példák

```
| ?- X in (10..20)/\ (\{15}), Y in 6..sup, Z #= X+Y.
   X in(10..14)\/(16..20), Y in 6..sup, Z in 16..sup ?

| ?- X in 10..20, X #\= 15, Y in {2}, Z #= X*Y.
   Y = 2, X in(10..14)\/(16..20), Z in 20..40 ?
```

A térképszínezési feladat SICStus-ban

```
| ?- use_module(library(clpfd)).
...
| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E.
            A in 2..3, B in 1..2,
            C in 1..3, D in 1..2, E in 2..3 ? ;
            no

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E,
     member(A, [1,2,3]). % címkézés, hivatalosan:
%    indomain(A).        % vagy:
%    labeling([], [A]).  % általánosan:
%    labeling([], [A,B,C,D,E]).
            A = 3, B = 2, C = 1, D = 1, E = 2 ? ;
            no

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= E, B #\= C, B #\= D, D #< E,
%    A #\= C, A #\= E, C #\= E helyett:
     all_distinct([A,C,E]).
     % Az ,,A, C, E különbözőek'' korlát okos
     % megvalósítása, globális kombinatorikai korláttal
            A = 3, B = 2, C = 1, D = 1, E = 2 ? ; no
```

Címkéző könyvtári eljárások — rövid előzetes

- `indomain(`*X*`)`: X-et a tartománya által megengedett értékkel helyettesíti, visszalépéskor felsorolja az összes értéket (növekedő sorrendben)
- `labeling(`*Opciók, Változók*`)`: A *Változók* lista minden elemét behelyettesíti, az *Opciók* lista által előírt módon.

CSP/CLP programok: klasszikus példa

Kódaritmetikai feladat: SEND+MORE=MONEY

A feladvány: Írjon a betűk helyébe számjegyeket (azonosak helyébe azonosakat, különbözőek helyébe különbözőeket), úgy hogy az egyenlőség igaz legyen. Szám elején nem lehet 0 számjegy.

```
send(SEND, MORE, MONEY) :-
  length(List, 8),
  domain(List, 0, 9),               % tartományok
  send(List, SEND, MORE, MONEY),    % korlátok
  labeling([], List).               % címkézés

send(List, SEND, MORE, MONEY) :-
  List= [S,E,N,D,M,O,R,Y],
  alldiff(List), S #\= 0, M#\= 0,
  SEND #=  1000*S+100*E+10*N+D,
  MORE #= 1000*M+100*O+10*R+E,
  MONEY #= 10000*M+1000*O+100*N+10*E+Y,
  SEND+MORE #= MONEY.

% alldiff(L): L elemei mind különbözőek (buta
% megvalósítás). Lényegében azonos a beépített
% all_different/1 kombinatorikai globális korláttal.
alldiff([]).
alldiff([X|Xs]) :- outof(X, Xs), alldiff(Xs).

outof(_, []).
outof(X, [Y|Ys]) :- X #\= Y, outof(X, Ys).
```

```
| ?- send(SEND, MORE, MONEY).
       MORE = 1085, SEND = 9567, MONEY = 10652 ? ; no
| ?- List=[S,E,N,D,M,O,R,Y], domain(List, 0, 9),
     send(List, SEND, MORE, MONEY).
       List = [9,E,N,D,1,0,R,Y],
       SEND in 9222..9866,
       MORE in 1022..1088,
       MONEY in 10244..10888,
       E in 2..8, N in 2..8, D in 2..8,
       R in 2..8, Y in 2..8 ? ; no
```


Szűkítési szintek

Informálisan, `r(X,Y)` bináris relációra

- Tartomány-szűkítés: `X` tartományából minden olyan `x` értéket elhagyunk, amelyhez nem található `Y` tartományában olyan `y` érték, amelyre `r(x,y)` fennáll. Hasonlóan szűkítjük `Y` tartományát. (Ez él-konzisztenciát eredményez.)
- Intervallum-szűkítési lépés: `X` tartományából elhagyjuk annak **alsó vagy felső** határát, ha ahhoz nem található **`Y` tartományának szélső értékei közé eső** olyan `y` érték, amelyre `r(x,y)` fennáll, és fordítva. Ezeket a lépéseket ismételjük, ameddig szükséges.

Példa

- Legyen
  - `r(X,Y):` $X = abs(Y)$.
  - `X` tartománya `0..5`
  - `Y` tartománya `{-1,1,3,4}`
- A tartomány-szűkítés elhagyja `X` tartományából a `0,2,5` értékeket, eredménye $X \in$ `{1,3,4}`.
- Az intervallum-szűkítés `X` tartományából csak az `5` értéket hagyja el, eredménye $X \in$ `0..4`.
- Az intervallum-szűkítés kétféle módon is gyengébb mint a tartomány-szűkítés:
  - csak a tartomány szélső értékeit hajlandó elhagyni, ezért nem hagyja el a `2` értéket;
  - a másik változó tartományában nem veszi figyelembe a „lukakat”, így a példában `Y` tartománya helyett annak *lefedő intervallumát*, azaz a `-1..4` intervallumot tekinti — ezért nem hagyja el `X`-ből a `0` értéket.
- Ugyanakkor az intervallum-szűkítés általában konstans idejű művelet, míg a tartomány-szűkítés ideje (és az eredmény mérete) függ a tartományok méretétől.

Szűkítési szintek — definíciók

Jelölések

- Legyen C egy n-változós korlát, s egy tár,
- $D(X, s)$ az X változó tartománya az s tárban,
- $D'(X, s) = min(D(X, s))..max(D(X, s))$, az X változó tartományát *lefedő* (legszűkebb) *intervallum*.

A szűkítési szintek definíciója

- Tartomány-szűkítés (domain consistency)
  C **tartomány-szűkítő** ha minden szűkítési lépés lefutása után az adott C korlát él-konzisztens, azaz bármelyik X_i változójához és annak tetszőleges $V_i \in D(X_i, s)$ megengedett értékéhez található a többi változónak olyan $V_j \in D(X_j, s)$ értéke ($j = 1, \ldots, i-1, i+1, \ldots, n$),hogy $C(V_1, \ldots V_n)$ fennálljon.
- Intervallum-szűkítés (interval consistency)
  C **intervallum-szűkítő** ha minden szűkítési lépés lefutása után igaz, hogy C bármelyik X_i változója esetén e változó tartományának mindkét **vég**pontjához (azaz a $V_i = min(D(X_i, s))$ illetve $V_i = max(D(X_i, s))$ értékekhez) található a többi változónak olyan $V_j \in D'(X_j, s)$ értéke ($j = 1, \ldots, i-1, i+1, \ldots, n$), hogy $C(V_1, \ldots V_n)$ fennálljon.

Megjegyzések

- A tartomány-szűkítés lokálisan (egy korlátra nézve) a lehető legjobb;
- **DE** mégha minden korlát tartomány-szűkítő, a megoldás nem garantálható, pl. `| ?- domain([X,Y,Z], 1, 2), X#\=Y, X#\=Z, Y#\=Z.`
- Egy CLP(FD) probléma megoldásának hatékonysága fokozható:
  - több korlát összefogását jelentő ún. globális korlátokkal, pl. `all_distinct(L)`: Az `L` lista csupa különböző elemből áll;
  - redundáns korlátok felvételével.

Garantált szűkítési szintek SICStusban

A SICStus által garantált szűkítési szintek

- A halmaz-korlátok (triviálisan) tartomány-szűkítők.
- A *lineáris* aritmetikai korlátok legalább intervallum-szűkítők.
- A nem-lineáris aritmetikai korlátokra nincs garantált szűkítési szint.
- Ha egy változó valamelyik határa végtelen (`inf` vagy `sup`), akkor a változót tartalmazó korlátokra nincs szűkítési garancia (bár az aritmetikai és halmaz-korlátok ilyenkor is szűkítenek).
- A később tárgyalandó korlátokra egyenként megadjuk majd a szűkítési szintet.

Példák

```
| ?- X in {4,9}, Y in {2,3}, Z #= X-Y.
       % intervallum-szűkítő:
       X in {4}\/{9}, Y in 2..3, Z in 1..7 ?

| ?- X in {4,9}, Y in {2,3}, plus(Y, Z, X).
       % plus(A, B, C): A+B=C tartomány-szűkítő módon
       X in {4}\/{9}, Y in 2..3, Z in(1..2)\/(6..7) ?

| ?- X in {4,9}, Y in {2}, /* azaz Y=2 */, Z #= X-Y.
       % tartomány-szűkítő:
       Y = 2, X in {4}\/{9}, Z in {2}\/{7} ?

| ?- X in {4,9}, Z #= X-Y, Y=2.
       % így csak intervallum-szűkítő!
       % vö. fordítási idejű korlát-kifejtés
       Y = 2, X in {4}\/{9}, Z in 2..7 ?

| ?-domain([X,Y], -10, 10), X*X+2*X+1 #= Y.
       % Ez nem interv.-szűkítő, Y<0 nem lehet!
       X in -4..4, Y in -7..10 ?

| ?- domain([X,Y], -10, 10), (X+1)*(X+1) #= Y.
       % garantáltan nem, de intervallum-szűkítő:
       X in -4..2, Y in 0..9 ?
```


Korlátok végrehajtása

A végrehajtás fázisai

- A korlát kifejtése elemi korlátokra (fordítási időben, lásd később)
  pl. `X*X #> Y` $\Rightarrow$ `X*X #= Z, Z #> Y`
- A korlát felvétele (posting):
  - azonnali végrehajtás (pl. `X #< 3`), vagy
  - démon létrehozása: első szűkítés elvégzése, újra-aktiválási feltételek meghatározása, a démon elaltatása.
- A démon aktiválása
  - szűkítés elvégzése,
  - döntés a folytatásról:
    - a démon lefut (ha a korlát már következménye a tárnak);
    - vagy a démon újra elalszik.

Elemi korlátok működése — példák

`A #\= B` (tartomány-szűkítő)

- Mikor **aktiválódik**? Ha vagy `A` vagy `B` konkrét értéket kap.
- Hogyan **szűkít**? A felvett értéket kihagyja a másik változó tartományából.
- Hogyan **folytatódik** a démon végrehajtása? A démon befejezi működését (lefut).

`A #< B` (tartomány-szűkítő)

- **Aktiválás**: ha `A` alsó határa (min `A`) vagy `B` felső határa (max `B`) változik
- **Szűkítés**: `A` tartományából kihagyja az $X \geq$ max `B` értékeket,
  `B` tartományából kihagyja az $Y \leq$ min `A` értékeket
- **Folytatás**: ha max `A` $<$ min `B`, akkor lefut, különben újra elalszik. (SICStusban: lefut, ha `A` vagy `B` behelyettesítődik.)

Korlátok végrehajtása (folyt.)

`all_distinct([`A_1`,...])` (tartomány-szűkítő)

- **Aktiválás**: ha bármelyik változó tartománya változik
- **Szűkítés**: (páros gráfokban maximális párosítást kereső algoritmus segítségével) minden olyan értéket elhagy, amelyek esetén a korlát nem állhat fenn. Példa:

```
| ?- A in 2..3, B in 2..3, C in 1..3,
     all_distinct([A,B,C]).

                  C = 1, A in 2..3, B in 2..3 ?
```

- **Folytatás**: ha már csak egy nem-konstans argumentuma van, akkor lefut, különben újra elalszik. (Jobb döntésnek tűnhet lefutni, ha a tartományok mind diszjunktak, de a SICStus nem így csinálja, valószínűleg nem éri meg.)

`X+Y #= T` (intervallum-szűkítő)

- **Aktiválás**: ha bármelyik változó alsó vagy felső határa változik
- **Szűkítés**: `T`-t szűkiti a `(min X+min Y)..(max X+max Y)` intervallumra, `X`-t szűkiti a `(min T-max Y)..(max T-min Y)` intervallumra, `Y`-t analóg módon szűkiti.
- **Folytatás**: ha (a szűkítés után) mindhárom változó konstans, akkor lefut különben újra elalszik.

Példa a szűkítések kölcsönhatására

```
| ?- domain([X,Y], 0, 100), X+Y #=10, X-Y #=4.
               X in 4..10, Y in 0..6 ?

| ?- domain([X,Y], 0, 100), X+Y #=10, X+2*Y #=14.
               X = 6, Y = 4 ?
```


Miért más a CLP(FD), mint a többi CLP rendszer?

A CLP könyvtárak összehasonlítása

	`clpq/r`	`clpb`	`clpfd`
Korlátok:	aritmetikai	logikai	aritmetikai. logikai, kombinatorikai,...
Egyszerű korlátok:	lineárisak	összes	*X* `in` *Halmaz*
Összetett korlátok végrehajtása:	várakozás, míg lineáris nem lesz	nincs ilyen	erősítés (szűkítés)
A tár konzisztenciájának biztosítása:	Gauss elimináció, szimplex	Bináris Döntési Diagrammok	triviális: *X* `in` *Halmaz*→ *Halmaz* nem üres
Az összes korlát konzisztenciájának biztosítása:	lineáris esetben automatikus	automatikus	csak címkézésen keresztül
Átlátszóság:	fekete doboz	fekete doboz	üveg-doboz
Kiterjeszthetőség:	nem	nem	igen

A CLP(FD) fő jellemzői

- A tár konzisztenciájának biztosítása triviális.
- A lényeg a démonok erősítő (szűkítő) működésében van.
- A démonok nem látják egymást, csak a táron keresztül hatnak egymásra.
- Globális korlátok: egyszerre több (akárhány) korlátot helyettesítenek, így erősebb szűkítést adnak (pl. `all_distinct`).
- A megoldás megléte általában csak a címkézéskor derül ki.

A CLP(FD) jellemzői — példák

```
| ?- domain([X,Y,Z], 1, 2), X #\= Y, X #\= Z, Y #\= Z.
                X in 1..2, Y in 1..2, Z in 1..2 ?

| ?- X #> Y, Y #> X.
                X in inf..sup, X in inf..sup ?

| ?- domain([X,Y], 1, 10), X #> Y, Y #> X.
                no

| ?- statistics(runtime,_),
     (   domain([X,Y], 1, 1000000),  X #> Y, Y #> X
     ;   statistics(runtime,[_,T])
     ).
                T = 3630 ?
```

A szűkítések nyomkövetése az FDBG könyvtár segítségével

```
| ?- use_module(library(fdbg)).
| ?- fdbg_on, fdbg_assign_name(X, x), fdbg_assign_name(Y, y),
     domain([X,Y], 1, 10), X #> Y, Y #> X.

domain([<x>,<y>], ==> x = inf..sup -> 1..10,
       1,10)          y = inf..sup -> 1..10
                      Constraint exited.

<x> #>= <y>+1     ==> x = 1..10 -> 2..10,   y = 1..10 -> 1..9

<x>+1 #=< <y>     ==> x = 2..10 -> 2..8,    y = 1..9 -> 3..9

<x> #>= <y>+1     ==> x = 2..8 -> 4..8,     y = 3..9 -> 3..7

<x>+1 #=< <y>     ==> x = 4..8 -> 4..6,     y = 3..7 -> 5..7

<x> #>= <y>+1     ==> x = 4..6 -> {6},      y = 5..7 -> {5}
                      Constraint exited.

2 #=< 0           ==> Constraint failed.
% Valójában a korlát <x>+1 #=< <y>, azaz 6+1 #=< 5
no
```


Klasszikus CSP/CLP programok: a „zebra” feladat

A feladvány

Egy utcában öt különböző színű ház van egymás mellett. A házakban különböző nemzetiségű és foglalkozású emberek laknak. Mindenki különböző háziállatot tart és más-más a kedvenc italuk is. A következőket tudjuk.

- Az angol a piros házban lakik.
- A festő japán.
- A norvég a balszélső házban lakik.
- A zöld ház a fehérnek jobboldali szomszédja.
- A diplomata a sárga házban lakik.
- A hegedűművész gyümölcslevet iszik.
- Az orvos szomszédja rókát tart.
- A spanyol kutyát tart.
- Az olasz a teát kedveli.
- A zöld házban lakó kávét iszik.
- A szobrász csigát tart.
- A tejet a középső házban kedvelik.
- A norvég a kék ház mellett lakik.
- A diplomata melletti házban lovat tartanak.

Kérdés: Kinek a háziállata a zebra?
(Lásd pl. `http://brownbuffalo.sourceforge.net/zebra.html`)

Modellezés

- változók meghatározása: egy-egy változó tartozik minden nemzetiséghez, háziállathoz, házszinhez, foglalkozáshoz és italhoz.
- változóértékek kódolása: A változó értéke annak a háznak a száma (balról számozva), amelynek lakóját, állatát, színét, stb. jelöli az adott változó.
- korlátok meghatározása:
  - az egyes változó-csoportok mind különböznek: `all_different/1` könyvtári korlát, pl. `all_different([Angol,Spanyol,Japán,Norvég,Olasz])`
  - két tulajdonság azonossága: egy `#=` korlát, pl. „Az angol a piros házban lakik.” ⇒ `Angol #= Piros`
  - két tulajdonság szomszédossága: házszámok különbsége 1, ill. 1 abszolút értékű, pl. „A norvég a kék ház mellett lakik” ⇒ `abs(Norvég-Kék)#=1`
  - A sorban egy konkrét ház megnevezése: egy számmal való egyenlőség, pl. „A tejet a középső házban kedvelik.” ⇒ `Tej #= 3`.

A „zebra” feladvány CLPFD megoldása

```
:- use_module(library(lists)).
:- use_module(library(clpfd)).

% ZOwner a zebra tulajdonosának nemzetisége, All az
% összes változó értéke a "Kié a zebra" feladványban.
zebra(ZOwner, All):-
  All = [England,Spain,Japan,Norway,Italy,
         Dog, Zebra, Fox, Snail, Horse,
         Green,Red,Yellow,Blue,White,
         Painter,Diplomat,Violinist,Doctor,Sculptor,
         Juice,Water,Tea,Coffee,Milk],
  domain(All, 1, 5),
  all_different([England,Spain,Japan,Norway,Italy]),
  all_different([Green,Red,Yellow,Blue,White]),
  all_different([Painter,Diplomat,Violinist,
                 Doctor,Sculptor]),
  all_different([Dog,Zebra,Fox,Snail,Horse]),
  all_different([Juice,Water,Tea,Coffee,Milk]),
  England #= Red,           Spain #= Dog,
  Japan #= Painter,         Italy #= Tea,
  Norway #= 1,              Green #= Coffee,
  Green #= White+1,         Sculptor #= Snail,
  Diplomat #= Yellow,       Milk #= 3,
  Violinist #= Juice,       nextto(Norway, Blue),
  nextto(Fox, Doctor),      nextto(Horse, Diplomat),
  labeling([], All),
  nth(N, [England,Spain,Japan,Norway,Italy], Zebra),
  nth(N, [english,spain,japan,norway,italy], ZOwner).

% A és B szomszédos számok.
nextto(A, B) :-  abs(A-B) #= 1.

| ?- zebra(ZOwner, All).
        All = [3,4,5,1,2,4,5,1,3,2|...],
        ZOwner = japan ? ; no
```


CSP/CLP programok: N vezér a sakktáblán

A feladvány

Egy N*N-es sakktáblán N vezért kell elhelyezni úgy, hogy egyik se üsse semelyik másikat, azaz ne legyen két vezér ugyanabban a sorban, ugyanabban az oszlopban, vagy ugyanazon átlós irányú vonal mentén.

Modellezés

- változók meghatározása: Minden vezérhöz egy változót rendelünk. Az X_i változó írja le az i. sorban levő vezér helyzetét.
- változóértékek kódolása: Az X_i változó azt az oszlopot jelöli, amelybe az i. sorban levő vezér kerül.
- korlátok meghatározása:
  - ne legyen két vezér egy sorban: nem szükséges külön korlát, mert a modellezés (változók jelentése) automatikusan biztosítja.
  - ne legyen két vezér egy oszlopban: X_i `#\=` X_j, minden $1 \leq i < j \leq N$ esetén.
  - minden átlós vonalban legfeljebb egy vezér legyen: bármely két vezér vízszintes és függőleges távolsága különbözzék: abs(X_i`-`X_j) `#\=`$j - i$, minden $1 \leq i < j \leq N$ esetén.
  - **Összegezve:** minden `X, Y` változópárra amelyek sortávolsága `I` (azaz `X =` X_i`, Y =` X_j`, I =` $abs(i - j)$) a következő három korlát fennállását kell biztosítani:
    `Y #\= X, Y #\= X-I, Y #\= X+I`
  - A fenti korlátok eljárásba foglalása:

```
% Az X és Y oszlopokban  I sortávolságra levő
% vezérek nem támadják egymást.
no_threat(X, Y, I) :-
        Y #\= X, Y #\= X-I, Y #\= X+I.
```

Egy bonyolultabb példa: mágikus sorozatok

Definíció: Egy $L = (x_0, \ldots, x_{n-1})$ sorozat *mágikus* ($x_i \in [0..n-1]$), ha L-ben az i szám pontosan x_i-szer fordul elő (minden $i \in [0..n-1]$-re).

Példa: n=4 esetén (1,2,1,0) és (2,0,2,0) mágikus sorozatok.

```
% Az L lista egy N hosszúságú mágikus sorozat.
magikus(N, L) :-
        length(L, N), N1 is N-1, domain(L, 0, N1),
        elofordulasok(L, 0, L),
        labeling([], L).            % most felesleges

% elofordulasok([E_i, E_{i+1}, ...], i, Sor): Sor-ban az i
% szám E_i-szer, az i+1 szám E_{i+1}-szer stb. fordul elő.
elofordulasok([], _, _).
elofordulasok([E|Ek], I, Sor) :-
        pontosan(I, Sor, E),
        J is I+1, elofordulasok(Ek, J, Sor).

% pontosan(I, L, E): Az I szám L-ben E-szer fordul elő.
pontosan(I, L, 0) :-     outof(I, L).
pontosan(I, [I|L], N) :-
        N #> 0, N1 #= N-1, pontosan(I, L, N1).
pontosan(I, [X|L], N) :-
        N #> 0, X #\= I,   pontosan(I, L, N).
```

Példafutás:

```
| ?- spy pontosan/3, magikus(4, L).
 +      1      1 Call: pontosan(0,[_A,_B,_C,_D],_A) ? s
?+      1      1 Exit: pontosan(0,[1,0,_C,_D],1) ? z
 +      2      1 Call: pontosan(1,[1,0,_C,_D],0) ? s
 +      2      1 Fail: pontosan(1,[1,0,_C,_D],0) ? z
 +      1      1 Redo: pontosan(0,[1,0,_C,_D],1) ? s
?+      1      1 Exit: pontosan(0,[2,0,0,_D],2) ? z
(...)
 +      4      1 Call: pontosan(2,[2,0,0,_D],0) ? s
 +      4      1 Fail: pontosan(2,[2,0,0,_D],0) ? z
(...)
?+      1      1 Exit: pontosan(0,[3,0,0,0],3) ? z
(...)
?+      1      1 Exit: pontosan(0,[2,0,_D,0],2) ?
```


Mágikus sorozatok: redundáns korlátok

Állítás: Ha az $L = (x_0, \ldots, x_{n-1})$ sorozat mágikus,
akkor $\sum_{i<n} x_i = n$, és $\sum_{i<n} i * x_i = n$.

Hatékonyabb változat, a fenti redundáns korlátokkal

```
% N=10 esetén kb. 50-szer gyorsabb az előző programnál!
magikus2(N, L) :-
    length(L, N), N1 is N-1, domain(L, 0, N1),
    osszege(L, S),              % Σ_{i∈ [1..N]} L_i = S
    szorzatosszege(L, 0, SP),   % Σ_{i∈ [0..N-1]} i*L_{i+1} = SP
    call(S #= N), call(SP #= N), % lásd a megjegyzést
    elofordulasok(L, 0, L).     % lásd az előző lapon
```

Megjegyzés

- Az aritmetikai beépített eljárások megengednek (aritmetikai) struktúrákat tartalmazó változókat, pl. `Kif = S1+S2, ..., Kif =:= 0`.
- CLPFD-ben ez nem megengedett: `Kif=S1+S2, ..., Kif #= 0` ⇒ `Hiba!` Ennek oka: a korlát-kifejtés csak betöltéskor történik meg.
- A megoldás a korlát-kifejtési fázis késleltetése: `Kif=S1+S2, ..., call(Kif #= 0)`.

Segédeljárások

```
% osszege(L, Ossz): Ossz = Σ_i L_i
osszege([], 0).
osszege([X|L], X+S) :- osszege(L, S).

% szorzatosszege(L, I, Ossz): Ossz = I*L_1 + (I+1)*L_2 + ...
szorzatosszege([], _, 0).
szorzatosszege([X|L], I, I*X+S) :-
    J is I+1, szorzatosszege(L, J, S).

| ?- magikus2(4, L).
% visszalépés nélkül adja ki az első megoldást!
 +      1      1 Call: pontosan(0,[_A,_B,_C,_D],_A) ?
(...)
?+      1      1 Exit: pontosan(0,[2,0,2,0],2) ? z
```


Reifikáció: korlátok tükrözése

Egy korlát tükrözése (reifikációja):

- a korlát igazságértékének „tükrözése” egy 0-1 értékű korlát-változóban;
- jelölése: C `#<=>` `B`, jelentése: `B` tartománya `0..1` és `B` csakkor `1`, ha C igaz;
- példa: `(X #>= 3) #<=> B` jelentése: `B` az `X` $\geq$ `3` egyenlőség igazságértéke.

Megjegyzések

- Az ún. formula-korlátok (az eddig ismertetett aritmetikai és halmaz-korlátok) mind tükrözhetőek.
- A globális korlátok (pl. `all_different/1, all_distinct/1`) nem tükrözhetőek.
- A tükrözött korlátok is „közönséges” korlátok, csak definíciójuk és végrehajtásuk módja speciális.
- Példa: a `0..5` tartományon a `(X #>= 3) #<=> B` korlát teljesen megegyezik a `B #= X/3` korláttal.

Tükrözött korlátok végrehajtása

- A C `<=>` `B` tükrözött korlát végrehajtása többféle szűkítést igényel:
  a. amikor `B`-ről kiderül valami (azaz behelyettesítődik): ha `B=1`, fel kell venni (*post*) a korlátot, ha `B=0`, fel kell venni a negáltját.
  b. amikor C-ről kiderül, hogy levezethető a tárból: `B=1` kell legyen
  c. amikor $\neg C$-ről kiderül, hogy levezethető a tárból: `B=0` kell legyen
- A fenti a., b. és c. szűkítések elvégzését három különböző démon végzi.
- A levezethetőség-vizsgálat (b. és c.) különböző „ambíciókkal”, különböző bonyolultsági szinteken végezhető el.

Reifikáció — példák

- Alappélda, csak `B` szűkül:

```
| ?- X#>3 #<=> B.                    ⇒ B in 0..1
```

- Ha `B` értéket kap, akkor a rendszer felveszi a korlátot ill. a negáltját:

```
| ?- X#>3 #<=> B, B = 1.             ⇒ X in 4..sup
| ?- X#>3 #<=> B, B = 0.             ⇒ X in inf..3
```

- Ha levezethető a korlát vagy negáltja, akkor `B` értéket kap.

```
| ?- X#>3 #<=> B, X in 15..sup.      ⇒ B = 1
| ?- X#>3 #<=> B, X in inf..0.       ⇒ B = 0
```

- Ha a tár megengedi a korlát és negáltja teljesülését is, akkor `B` nem kap értéket.

```
| ?- X#>3 #<=> B, X in 3..4.         ⇒ B in 0..1
```

- A rendszer kikövetkezteti, hogy az adott tárban `X` és `Y` távolsága legalább `1`:

```
| ?- abs(X-Y)#>1 #<=> B, X in 1..4, Y in 6..10.
                  ⇒ B = 1
```

- Bár a távolság-feltétel itt is fennáll, a rendszer nem veszi észre!

```
| ?- abs(X-Y)#>1 #<=> B, X in {1,5}, Y in {3,7}.
                  ⇒ B in 0..1
```

- Ennek itt az az oka, hogy az aritmetika nem tartomány-konzisztens.

```
| ?- D #= X-Y,
     AD #= abs(D), AD#>1 #<=> B,
     X in {1,5}, Y in {3,7}.
             ⇒ D in -6..2, AD in 0..6, B in 0..1

| ?- plus(Y, D, X),          ⇐ tartomány-konzisztens összegkorlát
     AD #= abs(D), AD#>1 #<=> B,
     X in {1,5}, Y in {3,7}.
             ⇒ D in {-6,-2,2}, AD in {2,6}, B = 1
```

Korlátok levezethetősége

A levezethetőség (entailment) felderítésének szintjei

- Tartomány-levezethetőség (domain-entailment):
  A C n-változós korlát **tartomány-levezethető** az s tárból, ha változóinak s-ben megengedett tetszőleges $V_j \in D(Xj, s)$ értékkombinációjára ($j = 1, \ldots, n$), $C(V_1, \ldots V_n)$ fennáll.
- Intervallum-levezethetőség (interval-entailment):
  C **intervallum-levezethető** s-ből, ha minden $V_j \in D'(Xj, s)$ értékkombinációra ($j = 1, \ldots, n$), $C(V_1, \ldots V_n)$ fennáll.

Megjegyzések

- Ha C intervallum-levezethető, akkor tartomány-levezethető is.
- A tartomány-levezethetőség vizsgálata általában bonyolultabb, mint az intervallum-levezethetőségé. Például a `X #\= Y` korlát:
  - tartomány-levezethető, ha `X` és `Y` tartományai diszjunktak (a tartomány méretével arányos költség) ;
  - intervallum-levezethető, ha `X` és `Y` tartományainak lefedő intervallumai diszjunktak (konstans költség).

A SICStus által garantált levezethetőségi szintek

- A tükrözött halmaz-korlátok kiderítik a tartomány-levezethetőséget.
- A tükrözött *lineáris* aritmetikai korlátok legalább az intervallum-levezethetőséget kiderítik.
- A tükrözött nem-lineáris aritmetikai korlátokra nincs garantált szint.

Példák

```
| ?- X in 1..4, X #< Y #<=> B, X+Y #=9.
    B = 1, X in 1..4, Y in 5..8 ?
| ?- X+Y #= Z #<=> B, X=1, Z=6, Y in 1..10, Y#\=5.
    X = 1, Z = 6, Y in(1..4)\/(6..10), B in 0..1 ?
    % X+Y #\= Z tartomány-, de nem interv.-levezethető!
```


Mágikus sorozatok (folyt.)

Tükrözést használó változat

```
magikus3(N, L) :-
        length(L, N),
        N1 is N-1, domain(L, 0, N1),
        osszege(L, S), call(S #= N),
        szorzatosszege(L, 0, SS), call(SS #= N),
        elofordulasok3(L, 0, L),
        labeling([], L). % most már kell a címkézés!

% A korábbi elofordulasok/3 másolata
elofordulasok3([], _, _).
elofordulasok3([E|Ek], I, Sor) :-
        pontosan3(I, Sor, E),
        J is I+1, elofordulasok3(Ek, J, Sor).

% pontosan3(I, L, E): L-ben az I E-szer fordul elő.
pontosan3(_, [], 0).
pontosan3(I, [X|L], N) :-
        X #= I #<=> B, N #= N1+B, pontosan3(I, L, N1).
```

A mágikus sorozat megoldásainak összehasonlítása

Az összes megoldás előállítási ideje másodpercben, 1 perc időkorláttal, Pentium III, 600 MHz processzoron („—” = időtúllépés).

variáns/adat	n=10	n=20	n=40	n=80	n=160	n=320
választós	13.90	—	—	—	—	—
választós+`osszege`	0.22	—	—	—	—	—
vál.+`szorzatosszege`	0.02	0.55	44.04	—	—	—
vál.+`ossz`+`szorzossz`	0.02	0.29	17.98	—	—	—
tükrözéses	0.05	1.07	24.02	—	—	—
tükrözéses+`osszege`	0.01	0.14	1.71	20.15	—	—
tükr.+`szorzatosszege`	0.01	0.04	0.18	0.94	4.75	25.77
tükr.+`ossz`+`szorzossz`	0.01	0.05	0.19	0.95	4.61	23.57

Logikai korlátok

Logikai korlát argumentuma lehet

- egy `B` változó, `B` automatikusan a `0..1` tartományra szűkül;
- egy tetszőleges tükrözhető aritmetikai- vagy halmazkorlát;
- egy tetszőleges logikai korlát.

A logikai korlátok (egyben függvényjelként is használhatók)

`#\ Q`	negáció	`op(710,  fy, #\).`
`P #/\ Q`	konjunkció	`op(720, yfx, #/\).`
`P #\ Q`	kizáró vagy	`op(730, yfx, #\).`
`P #\/ Q`	diszjunkció	`op(740, yfx, #\/).`
`P #=> Q`	implikáció	`op(750, xfy, #=>).`
`Q #<= P`	implikáció	`op(750, yfx, #<=).`
`P #<=> Q`	ekvivalencia	`op(760, yfx, #<=>).`

A tükrözött és logikai korlátok kapcsolata

- A korábban bevezetett tükrözési jelölés (C `<=> B`) a fenti logikaikorlát-fogalom speciális esete.
- De: a (C `<=> B`) alakú *elemi* korlát az, amire a logikai korlátok visszavezetődnek.
- Példa: `X#=4 #\/ Y#>6` $\longrightarrow$
  `X#=4#<=>B1, Y#>6#<=>B2, B1+B2 #>0`
- **Vigyázat!** A diszjunktív logikai korlátok gyengén szűkítenek, pl. egy n-tagú diszjunkció csak akkor tud szűkíteni, ha egy kivételével valamennyi tagjának a negáltja levezethetővé válik (a példában ha `X#\=4` vagy `Y#=<6` levezethető lesz).

Példa: lovagok, lókötők és normálisak

Egy szigeten minden bennszülött lovag, lókötő, vagy normális. A lovagok mindig igazat mondanak, a lókötők mindig hazudnak, a normális emberek pedig néha hazudnak, néha igazat mondanak. Kódolás: `normális` → `2`, `lovag` → `1`, `lókötő` → `0`.

```
:- use_module(library(clpfd)).
:- op(700, fy, nem).     :- op(900, yfx, vagy).
:- op(800, yfx, és).     :- op(950, xfy, mondja).

% A B bennszülött mondhatja az Áll állítást.
B mondja Áll :- értéke(B mondja Áll, 1).

% értéke(A, Érték): Az A állítás igazságértéke Érték.
értéke(X = Y, E) :-
    X in 0..2, Y in 0..2, E #<=> (X #= Y).
értéke(X mondja M, E) :-
    X in 0..2, értéke(M, E0),
    E #<=> (X #= 2 #\/ E0 #= X).
értéke(M1 és M2, E) :-
    értéke(M1, E1), értéke(M2, E2), E #<=> E1 #/\ E2.
értéke(M1 vagy M2, E) :-
    értéke(M1, E1), értéke(M2, E2), E #<=> E1 #\/ E2.
értéke(nem M, E) :-
        értéke(M, E0), E #<=> #\E0.

% http://www.math.wayne.edu/~boehm/Probweek2w99sol.htm
% We are given three people, A, B, C, one of whom is
% a knight, one a knave, and one a normal (but not
% necessarily in that order). They make the following
% statements.              A: I am normal
%                          B: A is right
%                          C: I am not normal
| ?- all_different([A,B,C]), A mondja A = 2,
     B mondja A = 2, C mondja nem C =2,
     labeling([], [A,B,C]).
                A = 0, B = 2, C = 1 ? ; no
```

Globális aritmetikai korlátok

Ezek a korlátok nem tükrözhetőek.

`scalar_product(Coeffs, Xs, Relop, Value[,Options])`
Igaz, ha a `Coeffs` és `Xs` listák skalárszorzata a `Relop` relációban van a `Value` értékkel, ahol `Relop` aritmetikai összehasonlító operátor (`#=, #<`, stb.). Alapértelmezésben intervallum-szűkítést biztosít, kivéve ha `Options = [consistency(domain)]`, amikor is tartomány-konzisztensen szűkít. `Coeffs` egészekből álló lista, `Xs` elemei és `Value` egészek vagy korlát változók lehetnek.
Megjegyzés: minden lineáris aritmetikai korlát átalakítható egy `scalar_product` hívássá.

`sum(Xs, Relop, Value)` Jelentése: Σ `Xs Relop Value`.
Ekvivalens a következővel: `scalar_product(Csupa1, Xs, Relop, Value)`, ahol `Csupa1` csupa `1` számból álló lista, `Xs`-sel azonos hosszú.

`knapsack(Coeffs, Xs, Value)`
Jelentése `Coeffs` és `Xs` skalárszorzata `Value`. Tartomány-konzisztenciát biztosít. Feltétel: Csak nem-negatív számok megengedettek, a változók véges tartományúak kell legyenek.

`minimum(Value, Xs), maximum(Value, Xs)`
Jelentése: az `Xs` lista elemeinek minimuma/maximuma `Value`.

Példa

```
send(List, SEND, MORE, MONEY) :-
        List= [S,E,N,D,M,O,R,Y],
        Pow10 = [1000,100,10,1],
        all_different(List), S #\= 0, M#\= 0,
        scalar_product(Pow10, [S,E,N,D], #=, SEND),
        %   SEND #=  1000*S+100*E+10*N+D,
        scalar_product(Pow10, [M,O,R,E], #=, MORE),
        %   MORE #= 1000*M+100*O+10*R+E,
        scalar_product([10000|Pow10], [M,O,N,E,Y],
                       #=, MONEY),
        %   MONEY #= 10000*M+1000*O+100*N+10*E+Y,
        SEND+MORE #= MONEY.
```

Ezzel befejeztük a halmaz-, aritmetikai, logikai és tükrözött korlátok ismertetését.

A formula-korlátok megvalósítása

Formula-korlátok

- Formula-korlátnak hívjuk az operátoros jelöléssel írt korlátot, azaz az eddig ismertetetteket, kivéve a globális aritmetikai korlátokat.
- A formula-korlátokat a rendszer nem könyvtári eljárással valósítja meg, hanem a Prolog `goal_expansion/5` kampójának segítségével.
- A kampó-eljárás *fordítási időben* a formula-korlátot, egy `scalar_product/4` korlátra, és/vagy nem-publikus elemi korlátokra fejti ki.
- A formula-korlátok kifejtése `call/1`-be ágyazással elhalasztható a korlát *futási időben* való felvételéig.

A legfontosabb elemi korlátok a `clpfd` modulban

- aritmetika: `'x+y=t'/3 'x*y=z'/3 'x/y=z'/3 'x mod y=z'/3 '|x|=y'/2 'max(x,y)=z'/3 'min(x,y)=z'/3`
- összehasonlítás: `'x=y'/2, 'x=<y'/2, 'x\\=y'/2` és tükrözött változataik: `'x Rel y'(X,Y,B)`, ahol `Rel` $\in$ { `= =< \=` } .
- halmaz-korlátok: `propagate_interval(X,Min,Max)` `prune_and_propagate(X,Halmaz)`
- logikai korlátok: `bool(Muvkod,X,Y,Z) % jelentése: X Muv Y = Z`
- optimalizálások: `'x*x=y'/2 'ax=t'/3 'ax+y=t'/4 'ax+by=t'/5 't+u=<c'/3 't=u+c'/3 't=<u+c'/3 't\\=u+c'/3 't>=c'/2` stb.

Az elemi korlátok szűkítési szintje

- **Definíció:** A C korlát **pont-szűkítő**, ha minden olyan tár esetén tartomány-szűkítő, amelyben C változói, legfeljebb egy kivételével be vannak helyettesítve. (Másképpen: ha minden ilyen tár esetén a korlát a behelyettesítetlen változót pontosan a C reláció által megengedett értékekre szűkíti.)
- Az elemi korlátok többsége pont-szűkítő (kivétel: `mod`).

Korlátok kifejtése

Példák (`clpfd` betöltése után)

```
| ?- use_module(library(clpfd)).
| ?- clpfd:goal_expansion(X*X+2*X+1 #= Y, _, user, G, []).
        G = clpfd:('x*x=y'(X,_A),
            scalar_product([1,-2,-1],[Y,X,_A],#=,1)) ?

| ?- clpfd:goal_expansion((X+1)*(X+1) #= Y, _, user, G, []).
        G = clpfd:('t=u+c'(_A,X,1),'x*x=y'(_A,Y)) ?

| ?- clpfd:goal_expansion(abs(X-Y)#>1, _, user, G, []).
        G = clpfd:('x+y=t'(Y,_A,X),
               '|x|=y'(_A,_B),'t>=c'(_B,2)) ?

| ?- clpfd:goal_expansion(X#=4 #\/ Y#>6, _, user, G, []).
        G = clpfd:'x=y'(X,4,_A),
            clpfd:'x=<y'(7,Y,_B),
            clpfd:bool(3,_A,_B,1) ?   % 3 a \/ kódja

| ?- clpfd:goal_expansion(X*X*X*X #= 16, _, user, G, []).
        G = clpfd:('x*x=y'(X,_A),'x*y=z'(_A,X,_B),
                      'x*y=z'(_B,X,16)) ?

| ?- clpfd:goal_expansion(X in {1,2}, _, user, G, []).
        G = clpfd:propagate_interval(X,1,2) ?

| ?- clpfd:goal_expansion(X in {1,2,5}, _, user, G, []).
        G = clpfd:prune_and_propagate(X,[[1|2],[5|5]]) ?
```

Megjegyzések

- Lineáris korlátok esetén a kifejtés megőrzi a pont- és intervallum-szűkítést.
- Általános esetben a kifejtés még a pont-szűkítést sem őrzi meg, pl `| ?- X in 0..10, X*X*X*X#=16.` $\longrightarrow$ `X in 1..4`

CLPFD segédeljárások

Statisztika

- `fd_statistics(Kulcs, Érték)`: A `Kulcs`-hoz tartozó számláló `Érték`-ét kiadja és lenullázza. Lehetséges kulcsok és számlált események:
  - `constraints` — korlát létrehozása;
  - `resumptions` — korlát felébresztése;
  - `entailments` — korlát (vagy negáltja) levezethetővé válásának észlelése;
  - `prunings` — tartomány szűkítése;
  - `backtracks` — a tár ellentmondásossá válása (Prolog meghiúsulások nem számítanak).
- `fd_statistics`: az összes számláló állását kiírja és lenullázza őket.

```
% Az N-vezér feladat összes megoldása Ss, Lab címkézéssel való
% végrehajtása Time msec-ig tart és Btrks FD visszalépést igényel.
run_queens(Lab, N, Ss, Time, Btrks) :-
        fd_statistics(backtracks, _), statistics(runtime, _),
        findall(Q, queens(Lab, N, Q), Ss),
        statistics(runtime, [_,Time]),
        fd_statistics(backtracks, Btrks).
```

Válaszok formája (a még le nem futott, alvó korlátok kiírása a válaszban)

- `clpfd:full_answer`: ez egy dinamikus kampó eljárás. Alaphelyzetben nincs egy klóza sem, tehát nem sikerül. Ez esetben a rendszer egy kérdésre való válaszoláskor csak a kérdésben előforduló változók tartományát írja ki, az alvó korlátokat nem. Ha felveszünk egy ilyen eljárást és az sikeresen fut le, akkor a válaszban az összes változó mellett kiírja még a le nem futott összes korlátot is.

```
| ?- domain([X,Y], 1, 10), X+Y#=5. ⇒ X in 1..4, Y in 1..4 ?
| ?- assert(clpfd:full_answer).    ⇒ yes
| ?- domain([X,Y], 1, 10), X+Y#=5. ⇒ clpfd:'t+u=c'(X,Y,5),
                                       X in 1..4, Y in 1..4 ?
| ?- X+Y #= Z #<=> B.     ⇒ clpfd:'t=u IND'(Z,_A)#<=>B,
                             clpfd:'x+y=t'(X,Y,_A), B in 0..1, ...
| ?- retract(clpfd:full_answer).   ⇒ yes
| ?- X+Y #= Z #<=> B.              ⇒ B in 0..1, ...
```

CLPFD segédeljárások (folyt.)

FD változók belső jellemzői

- Az FD változókról a könyvtár által tárolt információk lekérdezhetők.
- Ezek felhasználhatók a címkézésben, globális korlátok írásában ill. nyomkövetésben.
- **Vigyázat!** Félreértés veszélye! Minden más használat nagy eséllyel hibás.

FD változók felismerése

- `fd_var(V)`: `V` egy a clpfd könyvtár által ismert változó.

Tartományok pillanatnyi jellemzőinek lekérdezése

- `fd_min(X, Min)`: A `Min` paramétert egyesíti az `X` változó tartományának alsó határával (ez egy szám vagy `inf` lehet).
- `fd_max(X, Max)`: `Max` az `X` felső határa (szám vagy `sup`).
- `fd_size(X, Size)`: `Size` az `X` tartományának számossága (szám vagy `sup`).
- `fd_dom(X, Range)`: `Range` az `X` változó tartománya, *`KonstansTartomány`* formában
- `fd_set(X, Set)`: `Set` az `X` tartománya ún. FD-halmaz formában.
- `fd_degree(X, D)`: `D` az `X`-hez kapcsolódó korlátok száma.

Példák

```
| ?- X in (1..5)\/{9}, fd_min(X, Min), fd_max(X, Max),
     fd_size(X, Size).
          Min = 1, Max = 9, Size = 6, X in(1..5)\/{9} ?
| ?- X in (1..9)/\ \(6..8), fd_dom(X, Dom), fd_set(X, Set).
          Dom = (1..5)\/{9}, Set = [[1|5],[9|9]], X in ... ?
| ?- queens_nolab(8, [X|_]), fd_degree(X, Deg).
          Deg = 21, X in 1..8 ?         % 21 = 7*3
```


FD-halmazok

Az FD-halmaz fogalma, alapműveletei

- Az FD-halmaz formátum a tartományok belső ábrázolási formája.
- Absztrakt adattípusként használandó, alapműveletei:
  - `is_fdset(S)`: `S` egy korrekt FD-halmaz.
  - `empty_fdset(S)`: `S` az üres FD-halmaz.
  - `fdset_parts(S, Min, Max, Rest)`: Az `S` FD-halmaz áll egy `Min..Max` kezdő intervallumból és egy `Rest` maradék FD-halmazból, ahol `Rest` minden eleme nagyobb `Max+1`-nél. Egyaránt használható FD-halmaz szétszedésére és építésére.

```
| ?- X in (1..9) /\ \(6..8), fd_set(X, _S),
     fdset_parts(_S, Min1, Max1, _).
          Min1 = 1,
          Max1 = 5,
          X in(1..5)\/{9} ?
```

- Az FD-halmaz tényleges ábrázolása: `[Alsó|Felső]` alakú szeparált zárt intervallumok rendezett listája. (A ‘`.(_,_)`’ struktúra memóriaigénye 33%-kal kevesebb mint bármely más ‘`f(_,_)`’ struktúráé.)

```
| ?- X in (1..9) /\ \(6..8), fd_set(X, S).
          S = [[1|5],[9|9]],
          X in(1..5)\/{9} ?
```

- FD-halmaz is használató szűkítésre:
  - `X in_set Set`: Az `X` változót a `Set` FD-halmazzal szűkíti.
  - **Vigyázat!** Ha a korlát-felvételi fázisban egy változó tartományát egy másik tartományának függvényében szűkítünk, ezzel nem érhetünk el „démoni” szűkítő hatást, hiszen ez a szűkítés csak *egyszer* fut le. Az `in_set` eljárást csak globális korlátok ill. testreszabott címkézés megvalósítására célszerű használni.

FD-halmazok (folyt.)

FD-halmazokat kezelő további eljárások

- `fdset_singleton(Set, Elt)`: `Set` az egyetlen `Elt`-ből áll.
- `fdset_interval(Set, Min, Max)`: `Set` a `Min..Max` intervallum (oda-vissza használható).
- `empty_interval(Min, Max)`: `Min..Max` egy üres intervallum. Ekvivalens a `\+fdset_interval(_, Min, Max)` hívással.
- `fdset_union(Set1, Set2, Union)`: `Set1` és `Set2` uniója `Union`, `fdset_union(ListOfSets, Union)`: a `ListOfSets` lista elemeinek uniója `Union`.
- `fdset_intersection/[3,2]` : Két halmaz ill. egy listában megadott halmazok metszete.
- `fdset_complement/2`: Egy halmaz komplemense.
- `fdset_member(Elt, Set)`: `Elt` eleme a `Set` FD-halmaznak.
- `list_to_fdset(List, Set), fdset_to_list(Set, List)`: Számlista átalakítása halmazzá, és fordítva.
- `range_to_fdset(Range, Set), fdset_to_range(Set, Range)`: Konstans tartomány átalakítása halmazzá és viszont.

Példa

```
| ?- list_to_fdset([2,3,5,7], _FS1),
     fdset_complement(_FS1, _FS2),
          % _FS2 ↔ \{2,3,5,7}
     fdset_interval(_FS3, 0, sup),
          % _FS3 ↔ 0..sup
     fdset_intersection(_FS2, _FS3, FS),
          % FS ↔ (0..sup)/\ \{2,3,5,7}
     fdset_to_range(FS, Range),
     X in_set FS.

FS = [[0|1],[4|4],[6|6],[8|sup]],
Range = (0..1)\/{4}\/{6}\/(8..sup),
X in(0..1)\/{4}\/{6}\/(8..sup) ?
```


Címkézési (keresési) stratégiák

CSP programok szerkezete (*ismétlés!*)

- változók és tartományaik megadása,
- korlátok felvétele (lehetőleg választási pontok létrehozása nélkül),
- címkézés (keresés).

A címkézési fázis feladata

- Adott változók egy halmaza,
- ezeket a tartományaik által megengedett értékekre szisztematikusan be kell helyettesíteni
- (miközben a korlátok fel-felébrednek, és visszalépést okoznak a nem megengedett állapotokban).
- Mindezt a lehető leggyorsabban, a lehető legkevesebb visszalépéssel kell megoldani.

A keresés célja lehet

- **egyetlen** (tetszőleges) megoldás előállítása,
- az **összes** megoldás előállítása,
- a valamilyen szempontból **legjobb** megoldás előállítása.

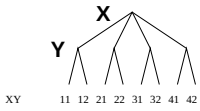
A keresési stratégia paraméterezési lehetőségei

- Milyen **sorrend**ben kezeljük az egyes változókat?
- Milyen **választási pont**ot hozunk létre?
- Milyen **irány**ban járjuk be a változó tartományát?

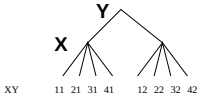
Keresési stratégiák — példák

Hogyan függ a keresési tér a változó-sorrendtől?

- `| ?- X in 1..4, Y in 1..2, indomain(X), indomain(Y).`



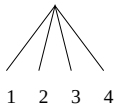
- `| ?- X in 1..4, Y in 1..2, indomain(Y), indomain(X).`



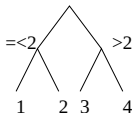
- A `first-fail` elv: a kisebb tartományú változót előbb címkézzük — kevesebb választási pont, remélhetően kisebb keresési tér.
- Példa feladatspecifikus sorrendre: az N vezér feladatban érdemes a középső sorokba tenni le először a vezéreket, mert ezek a többi változó tartományát jobban megszűrik, mint a szélsőkbe tettek.

Milyen szerkezetű keresési tereket hozhatunk létre?

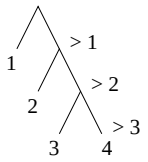
- felsorolás: `| ?- X in 1..4, labeling([enum], [X]).`



- kettévágás: `| ?- X in 1..4, labeling([bisect], [X]).`



- lépegetés: `| ?- X in 1..4, labeling([step], [X]).`



Címkéző eljárások

A címkézés alap-eljárása: `labeling(Opciók, VáltozóLista)`
A `VáltozóLista` minden elemét minden lehetséges módon behelyettesíti, az `Opciók` lista által előírt módon. Az alábbi csoportok mindegyikéből legfeljebb egy opció szerepelhet. Hibát jelez, ha a `VáltozóLista`-ban van nem korlátos tartományú változó. Ha az első négy csoport valamelyikéből nem szerepel opció, akkor a *`dőlt betűvel`* szedett alapértelmezés lép életbe.

1. a változó kiválasztása: *`leftmost`*, `min`, `max`, `ff`, `ffc`, `variable(Sel)`
2. a választási pont fajtája: *`step`*, `enum`, `bisect`, `value(Enum)`
3. a bejárási irány: *`up`*, `down`
4. a keresett megoldások: *`all`*, `minimize(X)`, `maximize(X)`
5. a gyűjtendő statisztikai adat: `assumptions(A)`
6. a balszélső ágtól való eltérés korlátozása: `discrepancy(D)`
7. időkorlát: `time_out(MSec,Result)`

A címkézés menete

a. Ha a változólista üres, akkor a címkézés sikeresen véget ér. Egyébként kiválasztunk belőle egy `X` elemet az 1. csoportbeli opció által előírt módon.

b. Ha `X` behelyettesített, akkor a változólistából elhagyjuk, és az **a.** pontra megyünk.

c. Egyébként az `X` változó tartományát felosztjuk két vagy több diszjunkt részre a 2. csoportbeli opció szerint (kivéve `value(Enum)` esetén, amikor is azonnal az **e.** pontra megyünk).

d. A tartományokat elrendezzük a 3. csoportbeli opció szerint.

e. Létrehozunk egy választási pontot, amelynek ágain sorra leszűkítjük az `X` változót a kiválasztott tartományokra.

f. Minden egyes ágon az `X` szűkítése értelemszerűen kiváltja a rá vonatkozó korlátok felébredését. Ha ez meghiúsulást okoz, akkor visszalépünk az **e.** pontra és ott a következő ágon folytatjuk.

g. Ha `X` most már behelyettesített, akkor elhagyjuk a változólistából. Ezután mindenképpen folytatjuk az **a.** pontnál.

h. Eközben értelemszerűen követjük a 4.-7. csoportbeli opciók előírásait is.

Speciális címkézési eljárás: `indomain(X)`

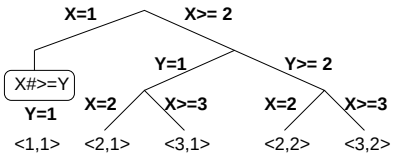
Ekvivalens a `labeling([enum], [X])` hívással.

A címkézés menete — példa

- A példa:
  `X in 1..3, Y in 1..2, X#>=Y, labeling([min], [X,Y]).`
- A `min` opció a legkisebb alsó határú változó kiválasztását írja elő.

```
| ?- fdbg_assign_name(X, x), fdbg_assign_name(Y, y),
     X in 1..3, Y in 1..2, X #>= Y, fdbg_on,
     labeling([min], [X,Y]).
% The clp(fd) debugger is switched on
Labeling [1, <x>]: starting in range 1..3.
Labeling [1, <x>]: step: <x> = 1
    <y>#=<1     y = 1..2 -> {1} Constraint exited.
                                                X = 1, Y = 1 ? ;
Labeling [1, <x>]: step: <x> >= 2
    <y>#=<<x>   y = 1..2, x = 2..3  Constraint exited.
    Labeling [6, <y>]: starting in range 1..2.
    Labeling [6, <y>]: step: <y> = 1
        Labeling [8, <x>]: starting in range 2..3.
        Labeling [8, <x>]: step: <x> = 2
                                                X = 2, Y = 1 ? ;
        Labeling [8, <x>]: step: <x> >= 3
                                                X = 3, Y = 1 ? ;
        Labeling [8, <x>]: failed.
    Labeling [6, <y>]: step: <y> >= 2
        Labeling [12, <x>]: starting in range 2..3.
        Labeling [12, <x>]: step: <x> = 2
                                                X = 2, Y = 2 ? ;
        Labeling [12, <x>]: step: <x> >= 3
                                                X = 3, Y = 2 ? ;
        Labeling [12, <x>]: failed.
    Labeling [6, <y>]: failed.
Labeling [1, <x>]: failed.
```

A keresési fa



Címkézési opciók

A címkézendő változó
A következő címkézendő változó kiválasztási szempontjai (ahol több szempont van, a későbbi csak akkor számít, ha a megelőző szempont(ok) szerint több azonos értékű van):

- `leftmost` (alapértelmezés) — legbaloldalibb;
- `min` — a legkisebb alsó határú; ha több ilyen van, közülük a legbaloldalibb;
- `max` — a legnagyobb felső határú; a legbaloldalibb;
- `ff` — („first-fail” elv): a legkisebb tartományú (vö. `fd_size`); a legbaloldalibb;
- `ffc` — a legkisebb tartományú; a legtöbb korlátban előforduló (vö. `fd_degree`); a legbaloldalibb;
- `variable(Sel)` — (meta-opció) `Sel` egy felhasználói eljárás, amely kiválasztja a következő címkézendő változót (lásd 87. oldal).

A választás fajtája
A kiválasztott `X` változó tartományát a következőképpen bonthatjuk fel:

- `step` (alapértelmezés) — `X #= B` és `X #\= B` közötti választás, ahol `B` az `X` tartományának alsó vagy felső határa (a bejárási iránytól függően);
- `enum` — többszörös választás `X` lehetséges értékei közül;
- `bisect` — `X #=< M` és `X #> M` közötti választás, ahol `M` az `X` tartományának középső eleme ($M = (min(X) + max(X))//2$);
- `value(Enum)` — (meta-opció) `Enum` egy eljárás, amelynek az a feladata, hogy leszűkítse `X` tartományát (lásd 88. oldal).

A bejárási irány
A tartomány bejárási iránya lehet:

- `up` (alapértelmezés) — alulról felfelé;
- `down` — felülről lefelé.

Címkézési opciók (folyt.)

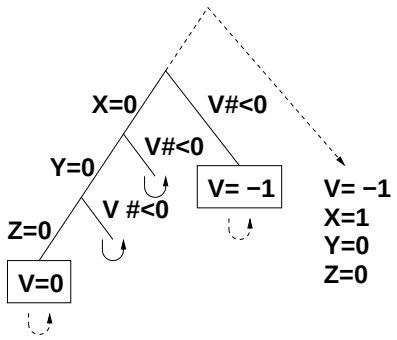
A keresett megoldások

- `all` (alapértelmezés) — visszalépéssel az összes megoldást felsorolja;
- `minimize(X)` ill. `maximize(X)` — egy, az `X`-re minimális ill. maximális értéket eredményező megoldást keres, branch-and-bound algoritmussal.

Példa szélsőérték keresésére

```
| ?- _L=[X,Y,Z], domain(_L, 0, 1),
     V#=Y+Z-X, labeling([minimize(V)], _L).

          V = -1, X = 1, Y = 0, Z = 0 ? ;
          no
```

A keresési fa a branch-and-bound algoritmussal



Címkézési opciók (folyt.)

Egyéb opciók

- Statisztika: `assumptions(K)` — egyesíti `K`-t a sikeres megoldáshoz vezető ágon levő változó-kiválasztások számával (ami lényegében a keresési fában a megoldáshoz vezető út hossza).
- A heurisztikától való eltérés korlátozása: `discrepancy(D)` (`D` adott szám)— csak olyan megoldásokat kérünk figyelembe venni, amelyekhez a keresési fában úgy jutunk el, hogy a legfeljebb `D`-szer választunk egy nem-legbaloldalibb ágat a választási pontokban. (Szemléletesen: a fa gyökerétől a megoldásig haladva legfeljebb `D`-szer kell megadni a jobbkéz-szabály szerinti elsőbbséget.)

  Az opció háttere az LDS (Limited Discrepancy Search) keresési módszer. Ebben feltételezzük, hogy a legbaloldalibb választások képviselik azt a heurisztikát, amivel nagy valószínűséggel eljuthatunk egy megoldáshoz. Mivel a heurisztika nem teljesen tökéletes, ezért valamennyi eltérést megengedünk, de az össz-eltérés-mennyiséget korlátozzuk.
- időkorlát: `time_out(MSec,Result)`. Ha `MSec` milliszekundum alatt lefut, `Result = success`, egyébként lelövi a cimkézést és `Result = time_out`. A minimize/maximize opciókkal jól működik együtt (ezek az opciók az addigi legjobb eredményt adják vissza).

Példák (vö. a 81. oldalon levő keresési fákkal):

```
assumptions(Select, As) :-
     X in 1..4,
     findall(A, labeling([Select,  assumptions(A)], [X]), As).

lds(Select, D, Xs) :-
     X in 1..4,
     findall(X, labeling([Select, discrepancy(D)], [X]), Xs).

| ?- assumptions(enum, As).          As = [1,1,1,1]
| ?- assumptions(bisect, As).        As = [2,2,2,2]
| ?- assumptions(step, As).          As = [1,2,3,3]

| ?- lds(enum, 1, Xs).               Xs = [1,2,3,4]
| ?- lds(bisect, 1, Xs).             Xs = [1,2,3]
| ?- lds(step, 1, Xs).               Xs = [1,2]
```


A címkézés testreszabása

`labeling/2` — a `variable(Sel)` meta-opció

- `variable(Sel)` — `Sel` egy eljárás, amely kiválasztja a következő címkézendő változót. `Sel(Vars,Selected,Rest)` alakban hívja meg a rendszer, ahol `Vars` a még címkézendő változók/számok listája.
- `Sel`-nek determinisztikusan sikerülnie kell egyesítve `Selected`-et a címkézendő *változóval* és `Rest`-et a maradékkal.
- `Sel` egy tetszőleges meghívható kifejezés lehet (`callable`, azaz név vagy struktúra). A három argumentumot a rendszer fűzi `Sel` argumentumlistájának végére.
- Például: ha a `Sel` opcióként a `mod:sel(Param)` kifejezést adjuk meg, akkor a rendszer a `mod:sel(Param,Vars,Selected,Rest)` eljáráshívást hajtja majd végre.

Példa a `variable` opció használatára

```
% A Vars-beli változók között Sel a Hol-adik,
% Rest a maradék.
valaszt(Hol, Vars, Sel, Rest) :-
        szur(Vars, Szurtek),
        length(Szurtek, Len), N is integer(Hol*Len),
        nth0(N, Szurtek, Sel, Rest).

% szur(Vk, Szk): A Vk-ban levő változók listája Szk.
szur([], []).
szur([V|Vk], Szk) :-       nonvar(V), !, szur(Vk, Szk).
szur([V|Vk], [V|Szk]) :-   szur(Vk, Szk).


queens([], 8, Qs).              → Qs = [1,5,8,6,3,7,2,4]
queens([variable(valaszt(0.5))], 8, Qs)
                                → Qs = [7,2,6,3,1,4,8,5]
queens([variable(valaszt(0.7))], 8, Qs)
                                → Qs = [5,7,2,6,3,1,4,8]
```


A címkézés testreszabása (folyt.)

`labeling/2` — a `value(Enum)` meta-opció

- `value(Enum)` — `Enum` egy eljárás, amelynek az a feladata, hogy leszűkítse `X` tartományát. Az eljárást a rendszer `Enum(X, Rest, BB0, BB)` alakban hívja meg, ahol `[X|Rest]` a még címkézendő változók listája.
- `Enum`-nak nemdeterminisztikusan le kell szűkítenie `X` tartományát az összes lehetséges módon, vö. a címkézés menetének leírását a 82. oldalon. (A `value` opció a **c.**, **d.** és **e.** lépések együttesét váltja ki.)
- Az első választásnál meg kell hívnia az `first_bound(BB0, BB)`, a későbbieknél a `later_bound(BB0, BB)` eljárást, a BB ill. LDS keresési algoritmusok kiszolgálására.
- `Enum`-nak egy meghívható kifejezésnek kell lennie. A négy argumentumot a rendszer fűzi `Enum` argumentumlistájának a végére.

Példa: belülről kifelé való érték-felsorolás

```
midout(X, _Rest, BB0, BB) :-
       fd_size(X, Size),
       Mid is (Size+1)//2,
       fd_set(X, Set),
       fdset_to_list(Set, L),
       nth1(Mid, L, MidElem),
       (   first_bound(BB0, BB), X = MidElem
       ;   later_bound(BB0, BB), X #\= MidElem
       ).

| ?- X in {1,3,12,19,120},
     labeling([value(midout)], [X]).
X = 12 ? ;
X = 3 ? ;
X = 19 ? ;
X = 1 ? ;
X = 120 ? ; no
```

A címkézés hatékonysága

A korábbi `queens` eljárás megoldásai 600 MHz Pentium III gépen.

Összes megoldás keresése

méret	n=8		n=10		n=12	
megoldások száma	92		724		14200	
címkézés	sec	btrk	sec	btrk	sec	btrk
`[step]`	0.07	324	1.06	5942	25.39	131K
`[enum]`	0.07	324	1.03	5942	24.84	131K
`[bisect]`	0.07	324	1.07	5942	26.04	131K
`[enum,min]`	0.08	462	1.31	8397	33.89	202K
`[enum,max]`	0.07	462	1.31	8397	33.89	202K
`[enum,ff]`	0.06	292	0.97	4992	21.57	101K
`[enum,ffc]`	0.06	292	1.04	4992	23.24	101K
`[enum,`*`midvar`*[1]`]`[2]	0.06	286	0.90	4560	20.11	88K

Első megoldás keresése

méret	n=16		n=18		n=20	
címkézés	sec	btrk	sec	btrk	sec	btrk
`[enum]`	0.43	1833	1.76	7436	9.01	37320
`[enum,min]`	0.52	2095	0.87	2595	1.39	3559
`[enum,max]`	0.61	3182	2.68	13917	16.06	83374
`[enum,ff]`	0.03	7	0.05	11	0.08	33
`[enum,ffc]`	0.03	7	0.05	11	0.09	33
`[enum,`*`midvar`*[1]`]`[2]	0.04	69	0.06	57	0.15	461
`[value(midout)`[2]`]`	0.04	3	0.05	4	0.09	38
`[value(midout)`[2]`,ffc]`	0.04	15	0.06	41	0.08	20

[1] *`midvar`* ≡ `variable(valaszt(0.5))`.
[2] Hatékonyabb statikusan (a címkézés előtt egyszer) elrendezni a változókat és az értékeket, lásd az `alt_queens/2` eljárást a `library('clpfd/examples/queens')` állományban.

Szélsőértékek ismételt hívással való előállítása

`minimize(Cél, X)` ill. `maximize(Cél, X)`
A `Cél` *ismételt hívásával* megkeresi az `X` változó minimális ill. maximális értékét.

A `minimize/2` eljárás definíciója

```
my_minimize(Goal, Var) :-
        findall(Goal-Var, (Goal -> true), [Best1-UB1]),
        minimize(Goal, Var, Best1, UB1).

% minimize(Goal, Var, BestSoFar, UB): Var is the minimal value < UB
% allowed by Goal, or, failing that, Goal = BestSoFar and Var = UB.
minimize(Goal, Var, _, UB) :- var(UB), !,  error.
                              % Goal does not instantiate Var
minimize(Goal, Var, _, UB) :-
        call(Var #< UB), %  csak a lenti nyomkövetés kedvéért
        findall(Goal-Var, (Goal -> true), [Best1-UB1]), !,
        minimize(Goal, Var, Best1, UB1).
minimize(Goal, Var, Goal, Var).
```

Magyarázatok a fenti definícióhoz

- `findall(Cél, (Cél->true), [EM])`: `EM` a `Cél` első megoldásának másolata.
- A keresési fa szerkezetétől függ, hogy a `minimize/2` vagy a `labeling([minimize...],...)` a hatékonyabb. Pl. a `minimize/2` a 85. oldalon levő fában elkerüli az `X`,`Y`-hoz tartozó választási pontok bejárását.

Példa a `my_minimize/2` használatára

```
p(L, V) :- L = [X,Y,Z], domain(L, 0, 1), V #= Y+Z-X.

| ?- spy [call/1,minimize/4,labeling/2].
| ?- p(L, V), my_minimize(labeling([], L), V).
 + 1 1 Call: lblg(user:[],[X,Y,Z]) ? z
?+ 1 1 Exit: lblg(user:[],[0,0,0]) ? z
 + 2 1 Call: minimize(lblg([],[X,Y,Z]),V,lblg([],[0,0,0]),0) ? z
 + 3 2 Call: call(user:(V#<0)) ? z
 + 3 2 Exit: call(user:(-1#<0)) ? z
 + 4 2 Call: lblg(user:[],[1,0,0]) ? z
 + 4 2 Exit: lblg(user:[],[1,0,0]) ? z
 + 5 2 Call: minimize(lblg([],[1,0,0]),-1,lblg([],[1,0,0]),-1) ? z
 + 6 3 Call: call(user:(-1#< -1)) ? z
 + 6 3 Fail: call(user:(-1#< -1)) ? z
 + 5 2 Exit: minimize(lblg([],[1,0,0]),-1,lblg([],[1,0,0]),-1) ? z
 + 2 1 Exit: minimize(lblg([],[1,0,0]),-1,lblg([],[0,0,0]),0) ? z
                L = [1,0,0], V = -1 ?
```


2. kis házi feladat: számkeresztrejtvény

A feladat

- Adott egy keresztrejtvény, amelyek egyes kockáiba $1..Max$ számokat kell elhelyezni (szokásosan $Max = 9$).
- A vízszintes és függőleges „szavak” meghatározásaként a benne levő számok összege van megadva.
- Egy szóban levő betűk (kockák) mind különböző értékkel kell bírjanak.

A keresztrejtvény Prolog ábrázolása:

- listák listájaként megadott mátrix;
- a fekete kockák helyén $F\backslash V$ alakú struktúrák vannak, ahol F és V az adott kockát követő függőleges ill. vízszintes szó összege, vagy `x`, ha nincs ott szó, *vagy egy egybetűs szó van*;
- a kitöltendő fehér kockákat (különböző) változók jelzik.

A megirandó Prolog eljárás és használata

```
% szamker(SzK, Max): SzK az 1..Max számokkal
% helyesen kitöltött számkeresztrejtvény.
% Megjegyzés: egyes sorban/oszlopban középen
% is lehet 'x'!

pelda(mini, [[x\ x,11\x,21\x, 8\x],
             [x\24,   _,   _,   _],
             [x\10,   _,   _,   _],
             [x\6,    _,   _, x\x]], 9).
```

	11	21	8
24	8	9	7
10	2	7	1
6	1	5	

```
| ?- pelda(mini, SzK, _Max), szamker(SzK, _Max).
               SzK = [[x\x, 11\x,21\x,8\x],
                      [x\24,8,   9,   7  ],
                      [x\10,2,   7,   1  ],
                      [x\6, 1,   5,   x\x]] ? ; no
```


Kombinatorikus (szimbolikus) korlátok

A kombinatorikus korlátok általános tulajdonságai

- A korlátok nem tükrözhetőek.
- Az argumentumaikban szereplő FD változók helyett mindig írható egész szám.

Értékek számolása

`count(Val, List, `*`Relop`*`, Count)`
Jelentése: a `Val` egész szám a `List` FD-változó-listában n-szer fordul elő, és fennáll az „*n* *`Relop`* `Count`” reláció. Itt `Count` FD változó, *`Relop`* a hat összehasonlító reláció egyike: `#=`, `#\=`, `#<` .... Tartomány-szűkítést biztosít.

`global_cardinality(Vars, Vals)`
`Vars` egy FD változókból álló lista, `Vals` pedig `I-K` alakú párokból álló lista, ahol `I` egy egész, `K` pedig egy FD változó. Mindegyik `I` érték csak egyszer fordulhat elő a `Vals` listában. Jelentése: A `Vars`-beli FD változók csak a megadott `I` értékeket vehetik fel, és minden egyes `I-K` párra igaz, hogy a `Vars` listában pontosan `K` darab `I` értékű elem van. Tartomány-szűkítést ad, ha `Vals` vagy `Vars` tömör, és még sok más speciális esetben.

Példa: mágikus sorozatok, újabb változatok

```
% Az L lista egy N hosszúságú mágikus sorozatot ír le.
magikus(N, L) :-
    length(L, N), N1 is N-1, domain(L, 0, N1),
       eloford(L, 0,          |      parok(L, 0, Pk, Egyhat),
               L, Egyhat),    |      global_cardinality(L, Pk),
    sum(L, #=, N), scalar_product(Egyhat, L, #=, N),
    labeling([], L).

% eloford([E_i, E_{i+1}, ...], i, Sor, Egyhat):
% Sor-ban az i szám E_i-szer, az i+1 szám E_{i+1}-szer stb.
% fordul elö. Egyhat az [i,(i+1),...] együttható-lista.
eloford([], _, _, []).
eloford([E|Ek], I, Sor, [I|EH]) :-
        count(I, Sor, #=, E),
        J is I+1, eloford(Ek, J, Sor, EH).

% parok([E_i, E_{i+1}, ...], i, Parok, Egyhat):
% Parok az [i-E_i, (i+1)-E_{i+1}, ...] párlista,
% Egyhat az [i,(i+1),...] együttható-lista.
parok([], _, [], []).
parok([E|Ek], I, [I-E|Pk], [I|EH]) :-
        J is I+1, parok(Ek, J, Pk, EH).
```