Nagyhatékonyságú logikai programozás

Jegyzetek a BME informatikus hallgatói számára

Szeredi Péter, Benkő Tamás

Számítástudományi
és Információelméleti Tanszék

IQSOFT Rt.

{szeredi,benko}@iqsoft.hu

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

Budapest 2000

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

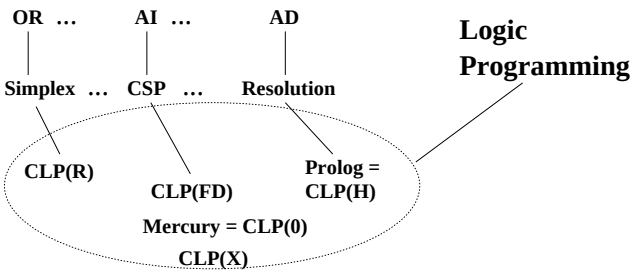
$\mathcal{X}$ = Q vagy R (a racionális vagy valós számok)
korlátok = lineáris egyenlőségek és egyenlőtlenségek
következtetési mechanizmus = Gauß elimináció és szimplex módszer

$\mathcal{X}$ = 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)

...

A CLP mint integrációs paradigma



„A CLP a lopás tudománya.”

(Mats Carlsson, SICS)

Példa: A SICStus `clpq` könyvtára

A Prologba ágyazás szintaxisa:

{*Korlát*} a *Korlát* felvétele

Példafutás

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

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

| ?- {X+Y+9<4*Z, 2*X=Y+2, 2*X+4*Z=36}.  % egyenlőtlenség is lehet
{X<29/5}, {Y= -2+2*X}, {Z=9-1/2*X} ?

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

| ?- {(Y+X)*(X+Y) = Y*Y+100*X}.         % így már nem...
clpq:{2*(X*Y)-100*X+X^2=0} ?

| ?- {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.   % így már igen...
X = -1/4, Y = -1/4 ?

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

A CLP($\mathcal{X}$) séma megvalósítási elvei

A korlát-tár

- *konzisztens* korlátok halmaza (konjunkciója)
- elemei az ún. primitív korlátok (a megengedett korlátok egy részhalmaza)
- az előremenő végrehajtás során a tár csak bővülhet (pontosabb lehet)
- ha a tár inkonzisztenssé válna, visszalépés történik (és a tár is visszaáll)
- a nem-primitív korlátok ágensként (démonként) várakoznak arra, hogy:
  a. primitív korláttá váljanak
  b. a tárat egy primitív korláttal bővíthessék (az ún. erősítés)

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 részletei

```
| ?- {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} ?
```


CLP rendszerek a nagyvilágban

Néhány implementáció

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

Kommerciális rendszerek (a fentiek között)

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

Mire használják a CLP rendszereket

Ipari erőforrás optimalizálás

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

Közlekedés, szállítás

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

Távközlés, elektronika

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

Egyéb

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

A SICStus clp(Q,R) kiterjesztései

A `clpq`/`clpr` könyvtárak

- Tartomány:
  - `clpr`: lebegőpontos számok
  - `clpq`: racionális számok
- Függvények:
  `+ - * / min max pow exp` (kétargumentumúak),
  `+ - abs sin cos tan` (egyargumentumúak).
- Constraint-relációk:
  `=, =:=, <, >, =<, >=, =\=`
- Primitív constraint-ek (constraint tár elemei):
  lineáris kifejezéseket tartalmazó relációk
- Constraint-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

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

Egy összetettebb példa: hiteltörlesztés

```
| ?- [user].
% 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).
| {user consulted, 20 msec 160 bytes}


| ?- 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(Constraint)` — `Constraint` 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, {Z=30*X+50*Y}, {X+1/2*Y=<8}, {X+3*Y=<15}, {X+2*Y=<11}
```

`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
```

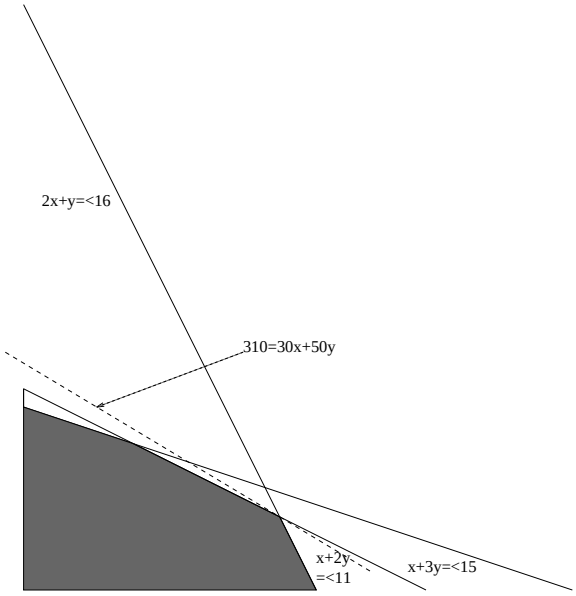
`bb_inf(Egészek, Kif, Inf)` — kiszámolja `Kif` infimumát, azzal a további feltétellel, hogy az Egészek listában levő minden változó egész (ún. "Mixed Integer Optimisation Problem").

```
| ?- {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-constraintben mint a `V2` változó.

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

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:

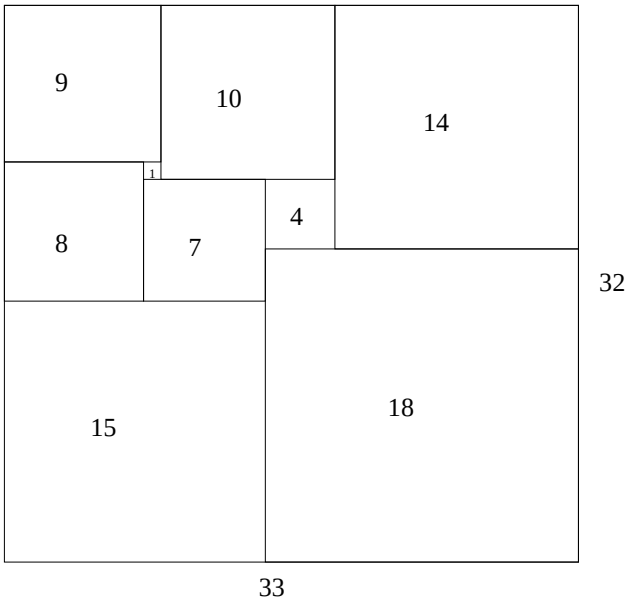
```
| ?- {X=0.5}, X=0.5.
no
| ?- {X=0.5}, X=1/2.
no
| ?- {X=0.5}, X=rat(2,4).
no
| ?- {X=0.5}, X=rat(1,2).
X = 1/2 ?
| ?- {X=5}, X=5.
no
| ?- {X=5}, X=rat(5,1).
X = 5 ?
```

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

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

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.
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 on
% HL horizontal line gives (vertical) line L.
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 }.

?- length(Ss, 9), filled_rectangle(Width, Ss).
    Ss = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32],
    Width = 33/32 ?
```


Nem teljesen tökéletes téglalapok

Az `outof` hívás kihagyásával kapott eredmények

```
W =   1, Ss = [  1]
W =   2, Ss = [  1,  1]
W = 3/2, Ss = [1/2,  1,1/2]
W = 3/2, Ss = [  1,1/2,1/2]
W =   3, Ss = [  1,  1,  1]
W = 4/3, Ss = [1/3,  1,1/3,1/3]
W =   1, Ss = [1/2,1/2,1/2,1/2]
W = 5/3, Ss = [2/3,  1,1/3,1/3]
W = 4/3, Ss = [  1,1/3,1/3,1/3]
W = 5/3, Ss = [  1,2/3,1/3,1/3]
W = 5/3, Ss = [1/3,1/3,  1,2/3]
W = 5/2, Ss = [1/2,  1,  1,1/2]
W = 5/3, Ss = [  1,1/3,1/3,2/3]
W = 5/2, Ss = [  1,1/2,  1,1/2]
W = 5/2, Ss = [  1,  1,1/2,1/2]
W =   4, Ss = [  1,  1,  1,  1]
W = 5/4, Ss = [1/4,  1,1/4,1/4,1/4]
W = 7/5, Ss = [2/5,  1,2/5,1/5,1/5]
W = 7/6, Ss = [1/2,2/3,1/2,1/3,1/3]
W = 6/5, Ss = [3/5,3/5,2/5,2/5,2/5]
W = 7/5, Ss = [2/5,  1,1/5,1/5,2/5]
W = 8/5, Ss = [3/5,  1,1/5,2/5,1/5]
W = 8/5, Ss = [3/5,  1,2/5,1/5,1/5]
W = 7/4, Ss = [3/4,  1,1/4,1/4,1/4]
W = 7/6, Ss = [2/3,1/2,1/2,1/3,1/3]
W = 5/4, Ss = [  1,1/4,1/4,1/4,1/4]
W = 7/5, Ss = [  1,2/5,2/5,1/5,1/5]
W = 7/5, Ss = [  1,2/5,1/5,1/5,2/5]
W = 8/5, Ss = [  1,3/5,1/5,2/5,1/5]
W = 8/5, Ss = [  1,3/5,2/5,1/5,1/5]
...
```


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 függvényeknek egy halmaza, pl. $+$, $-$, $*$, $\vee$, $\wedge$
- $\mathcal{R}$: $\mathcal{D}$-ben definiált relációknak (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: $\mathtt{P}$:- $\mathtt{G}_1, \ldots, \mathtt{G}_n$, ahol mindegyik $\mathtt{G}_i$ vagy cél, vagy korlát.
- deklaratív olvasat: $\mathtt{P}$ 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

Klózok procedurális olvasata

- `P :-` $\texttt{G}_1, \ldots, \texttt{G}_n$ megoldásához megoldandó $\texttt{G}_1, \ldots, \texttt{G}_n$.

Végrehajtási állapot

- $\langle$G, s$\rangle$
- G — cél/korlát sorozat
- s — korlát-tár: az eddig felhalmozott korlátok konjunkciója

Szükséges megkülönböztetés

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

Végrehajtási invariánsok

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

Végrehajtás vége

- $\langle \text{G}_e, \text{s}_e \rangle$, 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.

Következtetési lépések

Következtetési fajták

- rezolúció:
  $\langle$P & G, s$\rangle \Rightarrow \langle \texttt{G}_1$ & ... & $\texttt{G}_n$ & G, `P = P'` $\wedge$ s$\rangle$,
  ha a programban van egy `P' :-` $\texttt{G}_1, \ldots, \texttt{G}_n$ 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.

Követelmények

- inkrementalisság (az s tár konzisztenciáját ne bizonyítsa újra),
- a visszalépés támogatása,
- hatékonyság,
- teljesség.

A `clpb` könyvtár

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

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

- **Primitív constraintek** (constraint tár elemei): tetszőleges constraint (Boole-egyesítők formájában).
- **Constraint-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 *Kifejezést* a constraint-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.

Boole-egyesítés

A feladat:

- Adott `g` és `h` logikai kifejezések.
- Keressük a `g = h` egyenletet megoldó legáltalánosabb egyesítőt (mgu).
- Példa: mgu(`A+B`, 1) lehet `A = B*W + ~B`.
- 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 $f_{\texttt{X}}(1)$ az f-ből az `X=1`, $f_{\texttt{X}}(0)$ az `X=0` behelyettesítéssel kapott kifejezés.
- f `= 0` kielégíthetőségének szükséges feltétele: $f_{\texttt{X}}(1)$ `*` $f_{\texttt{X}}(0)$ `= 0`.
- Fejezzük ki `X`-et ezekkel úgy, hogy f `= 0` teljesülhessen!

$f_{\texttt{X}}(0)$	$f_{\texttt{X}}(1)$	`X`
0	0	`W`
0	1	0
1	0	1
1	1	`~W`

`X = ~W*`$f_{\texttt{X}}(0)$ `+ W*~`$f_{\texttt{X}}(1)$

- A legalsó sorban `X` választása önkényes de praktikus, mert egyszerűsíti az ábrázolást.

Bool-egyesítés (folyt.)

Az egyesítési algoritmus az f = 0 egyenlőségre

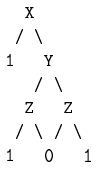
- Ha f-ben nincs változó, akkor azonosnak kell lennie 0-val (különben nem egyesíthető).
- Helyettesítsünk: X = W*~$f_X(1)$ + ~W*$f_X(0)$ (Boole-egyesítő)
- Folytassuk az egyesítést az $f_X(1)$ * $f_X(0)$ = 0 egyenlőségre.

Példák

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

Belső ábrázolás: BDD (Boolean/Binary Decision Diagrams)

X*Y # X*Z # 1 ábrázolása:



Egyszerű példák

```
| ?- use_module(library(clpb)).
{ ... loading ...}

| ?- 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) ?

| ?- taut(X, T).                              no

| ?- taut(x, T).                              T = 0 ?

| ?- taut(~x, T).                             T = 0 ?
```

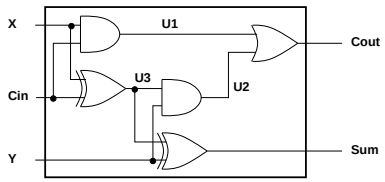
Megjegyzések

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

Példa: 1-bites összeadó

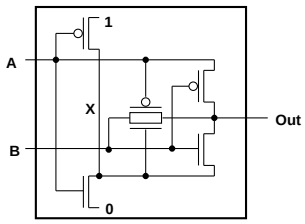
```
| ?- [user].
| adder(X, Y, Sum, Cin, Cout) :-
     sat(Sum =:= card([1,3],[X,Y,Cin])),
     sat(Cout =:= card([2-3],[X,Y,Cin])).
| {user consulted, 40 msec 576 bytes}

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
```


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_ex(1, Faults) :- fault(Faults, [1,1,0], [1,0]).
fault_ex(2, Faults) :- fault(Faults, [1,0,1], [0,0]).

| ?- fault_ex(I,L), labeling(L).

I = 1, L = [0,0,0,1,0] ? ;
I = 2, L = [1,0,0,0,0] ? ;
I = 2, 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)
```

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)).    :- use_module(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

A `clpfd` kiterjesztés alapelemei

- Tartomány: egészek (negatívak is!)
- Függvények (aritmetika):

  `+ - * / mod min max` (kétargumentumúak),

  `abs` (egyargumentumú).
- Constraint-relációk

  aritmetikaiak: `#<, #>, #=<, #>=, #= #\=`

  halmazműveletek: *X* `in` *Halmaz*, `domain([`*X*`,...], `*Min*, *Max*)

  Halmaz lehet: halmaz: {*Szám* ...}, intervallum: *Min* `..` *Max*, metszet: *Halmaz* `/\` *Halmaz*, únió: *Halmaz* `\/` *Halmaz*, komplemens: `\` *Halmaz*

  Min: *Szám* vagy `inf`

  Max: *Szám* vagy `sup`
- Primitív constraint-ek (constraint tár elemei): *X* `in` *Halmaz*
- Constraint-megoldó algoritmus:

  aritmetikaiak: constraint-szűkítés: ún. intervallum-konzisztencia

  halmazműveletek: teljes konzisztencia (ún. tartomány-konzisztencia)

Példa

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


Constraint feladatok megoldása

A clp(FD) programok tipikus felépítése

- deklaráljuk a változók tartományait
- állítsuk fel a constraint-eket
- helyettesítsük be a constraint-változókat (címkézés)

Címkézés

- `indomain(`*X*`)`: X-et a tartománya által megengedett értékkel helyettesíti (visszalépéssel felsorolja as összes értéket)
- `labeling(`*Opciók*, *Változók*`)`: A *Változók* lista minden elemét behelyettesíti, az *Opciók* lista által előírt módon, pl, `[]`: balról jobbra.

Példa: Kódaritmetika: SEND+MORE=MONEY

```
send(SEND,MORE,MONEY):-
  List= [S,E,N,D,M,O,R,Y], domain(List, 0, 9),
  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, labeling(List).

% labeling(Xs) ekvivalens a beépített labeling([],Xs) eljárással
labeling([]).
labeling([X|Xs]) :- indomain(X), labeling(Xs).

% Azonos a beépített all_different eljárással
alldiff([]).
alldiff([X|Xs]) :- outof(X, Xs), alldiff(Xs).

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

A zebra feladat

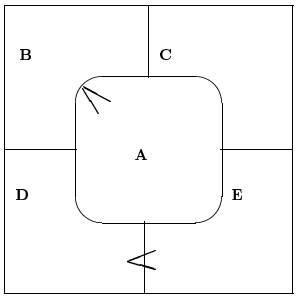
```
:- use_module(library(lists)).
:- use_module(library(clpfd)).

zebra(ZOwner, All):-
  All = [England,Spain,Japan,Norway,Italy,
         Green,Red,Yellow,Blue,White,
         Painter,Diplomat,Violinist,Doctor,Sculptor,
         Dog, Zebra, Fox, Snail, Horse,
         Juice,Water,Tea,Coffee,Milk],
  domain(All, 1, 5),
  alldiff([England,Spain,Japan,Norway,Italy]),
  alldiff([Green,Red,Yellow,Blue,White]),
  alldiff([Painter,Diplomat,Violinist,Doctor,Sculptor]),
  alldiff([Dog,Zebra,Fox,Snail,Horse]),
  alldiff([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([ff], All),
  nth(N, [England,Spain,Japan,Norway,Italy], Zebra),
  nth(N, [english,spain,japan,norway,italy], ZOwner).

nextto(A, B) :-  abs(A-B) #= 1.
```


Véges tartományú korlátok háttere

CSP —Korlát-Kielégítési Problémák
(Constraint Satisfaction Problems)

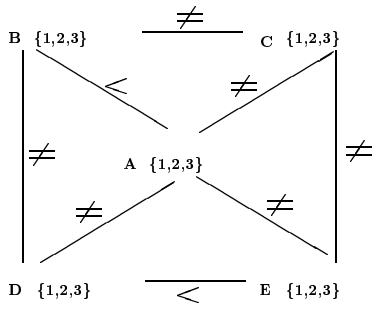
Példa

Az alábbi térkép kiszínezése három színnel, azaz az 1, 2, 3 számok beírása úgy, hogy a szomszédos mezők különböző értéket kapjanak és a beírt relációk is teljesüljenek.



Korlát-Kielégítési Problémák (folyt.)

Az előbbi térképnek megfelelő korlát-gráf:



A KKP megoldás lépései

- a változók tartományainak felvétele
- a korlátok felállítása
- a tartományok szűkítése
- címkézés — visszalépéses keresés

A térképszinezési példa 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,
     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),
     all_distinct([A,B,C]), all_distinct([A,C,E]),
     A #> B, E #> D.

A = 3, B = 2, C = 1, D = 1, E = 2 ? ;

no
| ?-
```

CLP(FD) — A korlát-tár működése

A CLP(FD) korlát-tár

- primitiv korlátok: X `in` *Tartomány*, pl. `X in 2..3 \/ 5`
- A CLP(FD) tár nem más, mint egy hozzárendelés a változók és lehetséges értékek között
- Új primitív korlát felvétele: egy változó tartományának szűkítése
- A felhasználói korlátok többsége ágens lesz: szűkít, elalszik, aktiválódik, szűkít, ...
- Az ágenseket a korlátbeli változók tartományának változása aktiválja

Példa: `A #\= B`

- Aktiválás: ha vagy `A` vagy `B` konkrét értékre helyettesítődik
- Szűkítés: az érték kihagyása a másik változó tartományából
- Folytatás: az ágens befelyezi működését

Példa: `A #< B`

- 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

N királynő a sakktáblán

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

% A Qs lista N királynő biztonságos elhelyezését
% mutatja egy N*N-es sakktáblán. Ha a lista
% i. eleme j, akkor az i. királynőt az i. sor j.
% oszlopába kell helyezni.
queens(N, Qs):-
        length(Qs, N), domain(Qs, 1, N),
        safe(Qs),
        labeling([ff],Qs).          % first-fail elv

% safe(Qs): A Qs lista a királynők biztonságos
% elhelyezését írja le.
safe([]).
safe([Q|Qs]):-
        no_attack(Qs, Q, 1),
        safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt
% királynők egyike sem támadja a Q oszlopban levő
% kiralynőt, feltéve hogy Q és Qs távolsága I.
no_attack([],_,_).
no_attack([X|Xs], Y, I):-
      no_threat(X, Y, I),
      J is I+1, no_attack(Xs, Y, J).

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


A constraint-ek működése

A constraint erősítés fajtái

Legyen C egy n-változós constraint; jelöljük *D(X,s)*-sel az X változó tartományát az s constraint-tárban.

- Tartomány-konzisztencia (domain consistency).

  C tartomány-konzisztens s-re nézve, ha bármelyik *Xi* változójához és annak tetszőleges *Vi* ∈ *D(Xi,s)* megengedett értékéhez található a többi változónak olyan *Vj* ∈ *D(Xj,s)* értéke (j = *1, ..., i-1,i+1, ..., n*), hogy *C(X1, ...Xn)* fennálljon.

  Jelöljük *D'(X,s)*-sel a *min(D(X,s)) .. max(D(X,s))* intervallumot.

- Intervallum-konzisztencia (interval consistency).

  C intervallum-konzisztens s-re nézve, ha bármelyik *Xi* változójához és tartománya mindkét végpontjához, azaz a *Vi = min(D(X,s))* és *Vi = max(D(X,s))* értékekhez található a többi változónak olyan *Vj* ∈ *D'(Xj,s)* értéke (j = *1, ..., i-1,i+1, ..., n*), hogy *C(X1, ...Xn)* fennálljon.

- Az aritmetikai constraint-ekre a SICStus clpfd könyvtár csak intervallum konzisztenciát garantál, de bizonyos esetekben tartomány-konzisztenciát is biztosít.

Példák

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


Példa: mágikus sorozatok

- 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([Ei, ...], i, Sor): Sor-ban i, i+1, ...
% előfordulásainak száma Ei, Ei+1, ...
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 pontosan 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).

% Hatékonyság-fokozás redundáns (surrogate) constraint-ekkel.
magikus1(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),
        elofordulasok(L, 0, L).

osszege([], 0).
osszege([X|L], X+S) :- osszege(L, S).

szorzatosszege([], _, 0).
szorzatosszege([X|L], I, I*X+S) :-
        J is I+1, szorzatosszege(L, J, S).
```

Reifikáció

Gondok a pontosan/3 eljárás előbbi megvalósításával

- harmadik argumentumának értéket ad, nem kezeli constraint-változóként,
- választási pontokat hoz létre

Egy constraint reifikációja

- A constraint igazságértékének "tükrözése" egy 0-1 értékű constraint-változóban.
- Példa: `(X #= 3) #<=> B`: `B` az `X=3` egyenlőség igazságértéke.
- Az aritmetikai és halmaz-constraintek mind reifikálhatóak

Reifikált constraint-ek végrehajtása

- egy démon figyeli, hogy a constraint-tárból levezethető-e, ha igen, `B=1`;
- egy démon figyeli, hogy levezethető-e a negáltja, ha igen, `B=0`;
- egy harmadik démon figyeli, hogy `B` értéket kap-e, ha `B=1`, felveszi a constraint-t, ha `B=0`, felveszi a negáltját a constraint-tárba.

Constraint-levezethetőség (entailment)

- **Tartomány-levezethetőség**: *C tartomány-levezethető s*-ből, ha Xi változói minden megengedett *Vj ∈ D(Xj,s)* értékére *(j = 1, ..., n)*, *C(X1, ...Xn)* fennáll.
- **Intervallum-levezethetőség**: *C intervallum-levezethető s*-ből, ha minden *Vj ∈ D'(Xj,s)* értékre *(j = 1, ..., n)*, *C(X1, ...Xn)* fennáll.

A reifikált aritmetikai constraint-ek intervallum-levezethetőséget, a reifikált halmaz-constraintek tartomány-levezethetőséget garantálnak.

Példák tüközött constraint-ekre

```
| ?- 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 ?
```

`pontosan/3` tükrözött constraint-ekkel

```
pontosan(_, [], 0).
pontosan(I, [X|L], N) :-
       X #= I #<=> B, N #= N1+B, pontosan(I, L, N1).
```

A tükrözés általánosítása: logikai müveletek constraint-ekkel

- `#\ Q`: `Q` negáltja; `P #/\ Q`: konjunkció; `P #\/ Q`: diszjunkció; `P #\ Q`: kizáró vagy; `P #=> Q` vagy `Q #<= P`: implikáció; `P #<=> Q`: ekvivalencia.
- `P` és `Q` tetszőleges reifikálható constraint (pl aritmetikai), vagy 0-1 értékű constraint-változó
- a logikai müveletek reifikált és aritmetikai constraint-ekre vezethetők vissza, pl: `X#=4 #\/ Y#>6` ⟶ `X#=4#<=>B1, Y#>6#<=>B2, B1+B2 #>0`

A reifikáció hatékonysága: mágikus sorozatok (586/133Mhz)

variáns/adat	n=8	n=10	n=12	n=14	n=20	n=30
választós	10.17	227.96				
választós+osszege	1.35	6.26	28.69	127.29		
vál.+szorzatosszege	0.27	0.66	1.32	2.54		
vál.+ossz.+szorzossz.	0.20	0.48	1.02	1.90	10.59	108.94
reifikálós	1.20	2.76				
reifikálós+osszege	0.47	0.95	1.73	2.54		
reif.+szorzatosszege	0.20	0.40	0.54	0.72		
reif.+ossz.+szorzossz.	0.25	0.43	0.62	0.82	1.89	4.83

Constraintek feldolgozása

A feldolgozás fázisai

- A constraint kifejtése elemi constraintekre
  - alapértelmezés: fordítási időben;
  - `call/1`-be ágyazással elhalasztható a constraint végrehajtásáig.
  - kisérletezés: `goal_expansion(`*Constraint*`, user, Kifejtés)`
- A constraint végrehajtása (posting)
  - megállapítjuk mikor kell felébreszteni; (a változói alsó, felső határának változásakor, bármilyen változáskor vagy értékadáskor);
  - felébresztjük a constraintet.
- A constraint felébresztése = constraint-erősítés

Elemi constraintek

- aritm.:`'x+y=t'/3 'x*y=z'/3 'x/y=z'/3 'x mod y=z'/3 '|x|=y'/2`
- összehasonlítás: `'x=y'/2, 'x=<y'/2, 'x\=y'/2` és reifikált változataik: `iff_aux('x` *Rel* `y'(X,Y), B)`, ahol *Rel* a `= =< \=` jelek valamelyike.
- 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.
- Példák:
  `abs(X-Y)#>1` ⟶ `'x+y=t'(Y,A,X),'|x|=y'(A,B),'t>=c'(B,2)`
  `(X#<3)#<=>B` ⟶ `iff_aux('x=<y'(X,2),B)`
  `X*X+Y*Y#=<16` ⟶ `'x*x=y'(X,A),'x*x=y'(Y,B),'t+u=<c'(A,B,16)`

Constraintek kifejtése

- lineáris kifejezés esetén megmarad az intervallum-konzisztencia;
- általános esetben nem, pl `X in 0..10, X*X*X*X#=16.` ⟶ `X in 1..4`
- de sokszor a nem-lineárisra is jó, pl. prímszámok ellenőrzésére: `domain([X,Y], 0, 5000), X-Y#>1, X*X-Y*Y#=6211.` ⟶ `no`

Constraint erősítés ($< C, s > \longrightarrow < C', s \wedge c >$ ahol $s \models C \equiv C' \wedge c$); használt speciális esetei:

- ha $s \wedge c$ inkonzisztens, akkor visszalépünk;
- ha $C' =$ `true`, a constraint levezethetővé vált, azaz sikeresen lefutott;
- ha $C' \neq C$, akkor C lefutott, helyette végrehajtjuk C'-t, pl. reifikációnál, vagy `max(X,Y) #= Z` ⟶ `X #= Z`, ha `Y #=< X`;
- ha $C' = C$, akkor (újra) elaltatjuk a constraintet.

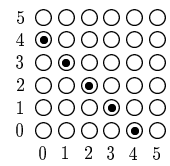
Constraint erősítés megvalósítása

- cél: az adott constraint és egy adott tár esetén kiszűrni azokat az értékeket, amelyekre a constraint nem teljesülhet.
- közvetlen (nem reifikált) constraintek esetén képezzük a constraint és a tár metszetét, majd:
  - tartomány-konzisztencia esetén: a változó-tartományok minden pontját próbáljuk szűrni, pl. `abs(X-Y)#>1, Y in 2..4` → `X#\=3`;
  - intervallum-konzisztencia biztosításakor: a tartomány végpontjait próbáljuk kiszűrni, a többi változó tartományát intervallumnak tekintve (ha sikerül, ismételjük);
- reifikált constraintek esetén:
  - ha a logikai változó értéket kap, a kérdezett constraint vagy negáltja lép a reifikált constraint helyébe;
  - vizsgáljuk, hogy a kérdezett constraint vagy negáltja a tárból levezethetővé válik-e:
    - tartomány-levezethetőség vizsgálata: ha minden a tár által megengedett behelyettesítésre igaz a constraint, pl. `Y in {0,5}, X in 2..3` tárból `abs(X-Y)#>1` tartomány-levezethető;
    - intervallum-levezethetőség vizsgálata: ha a tárban minden tartományt intervallummá általánosítunk, akkor is mindig igaz, pl `abs(X-Y)#>1` a fenti tárból nem intervallum-levezethető.

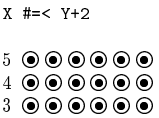
    Ha a kérdezett constraint vagy a negáltja levezethetővé válik, a logikai változó szűkül: a 0 vagy 1 értéket kapja.

`domain([X,Y], 0, 5), C(X,Y), X#=<Y+2, X#>2, Y in {0,4,5}`

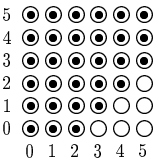
`C(X,Y):`

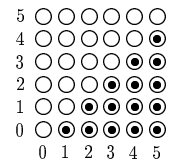


`X + Y #= 4`

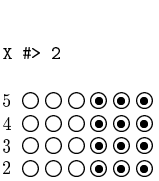


`X #=< Y+2`



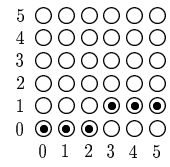


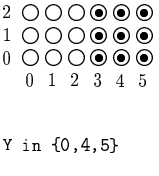
`X #> Y`



`X #> 2`

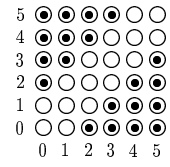
`Y #= X/3`
vagy
`X#>=3 #<=> Y`

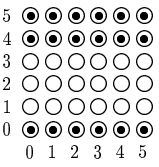


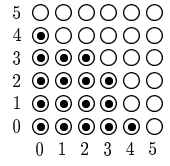


`Y in {0,4,5}`

`abs(X-Y)#>1`
vagy
`X-Y#=Z #∧`
`Z #>1 #∨`
`Z #< -1`







`X*X+Y*Y#=<16`

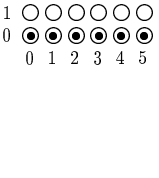
Constraint tárak

`X+Y #= 4, X#=<Y+2, X#>2`

`X #> Y, X#=<Y+2, X#>2, Y in {0,4,5}`

`X#>=3 #<=> Y, X#=<Y+2, X#>2`

`abs(X-Y)#>1, X#=<Y+2, X#>2, Y in {0,4,5}`



`X*X+Y*Y#=<16, X#=<Y+2, X#>2`

További aritmetikai constraintek

Ezek a constraintek nem reifikálhatóak.

`scalar_product(Coeffs, Xs, Relop, Value)`
Igaz, ha `Coeffs * Xs Relop Value`, ahol `Relop` aritmetikai összehasonlító operátor (`#=`, `#<`, stb.), `*` skalárszorzást jelöl, `Coeffs` egészekből álló lista, `Xs` elemei és `Value` egészek vagy constraint változók.
Intervallum-konzisztenciát biztosít.

`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.

Kombinatorikus (szimbolikus) constraintek

Ezek a constraintek nem reifikálhatóak.
`count(Val, List, Relop, Count)`
Jelentése: a `Val` egész szám `List`-ban való előfordulásainak száma n, és n `Relop Count`. (V. ö. a korábbi `pontosan/3` predikátummal.)

Példa:

```
magikus2(N, L) :-
        length(L, N), N1 is N-1, domain(L, 0, N1),
        elofordulasok(L, 0, L, Egyhat),
        sum(L, #=, N),
        scalar_product(Egyhat, L, #=, N),
        labeling([], L).

elofordulasok([], _, _, []).
elofordulasok([E|Ek], I, Sor, [I|EH]) :-
        count(I, Sor, #=, E),
        J is I+1, elofordulasok(Ek, J, Sor, EH).
```


Kombinatorikus constraintek (folyt.)

`element(X, CLista, Y)`
Jelentése: `CLista` `X`-edik eleme `Y`, ahol `X` és `Y` constraint-változók (vagy egészek), `CLista` egészekből vagy constraint-változókból álló lista.
Tartomány-konzisztenciát biztosít. Példák:

```
| ?- element(X, [0,1,2,3,4], Y), X in {2,5}.  % Y #= X-1
                       X in{2}\/{5}, Y in{1}\/{4} ?

% X #= c #<=> B megvalósítása, 1 =< X,c =< 6 esetére.
beq(X, C, B) :-
  X in 1..6, call(I #= X+6-C),
  element(I, [0,0,0,0,0,1,0,0,0,0,0], B).
```

`relation(X, Hozzárendelés, Y)`
Itt `X` és `Y` constraint-változók (vagy egészek), `Hozzárendelés` *egész* - *Halmaz* alakú párok listája (ahol mindegyik *egész* csak egyszer fordulhat elő).
Jelentése: `Hozzárendelés` tartalmaz egy `X`-`Halmaz` párt, ahol `Y` eleme a `Halmaz`-nak. Tetszőleges bináris reláció definiálására használható.
Tartomány-konzisztenciát biztosít. Példa:

```
'abs(x-y)>1'(X,Y) :-
  relation(X, [0-(2..5), 1-(3..5), 2-{0,4,5},
               3-{0,1,5}, 4-(0..2), 5-(0..3)], Y).

'abs(x-y)>1'(X,Y), X in 2..3. → Y in(0..1)∨(4..5)
```

`all_different(Változók)`
A Változók páronként különbözőek. Megvalósítása nem teljes, a korábbi `alldiff` eljárással azonos.

`all_distinct(Változók)`
Jelentése ugyanaz, mint `all_different`, de teljes (tartomány-konzisztenciát biztosít) . Például:

```
| ?- domain([X,Y,Z], 1, 2), all_distinct([X,Y,Z]).
no
```

Kombinatorikus constraintek (folyt.)

`assignment(Xs, Ys)`
`Xs` és `Ys` egészekből vagy constraint-változókból alkotott azonos (`n` hosszúságú listák. Teljesül, ha `X[i]` és `Y[i]` mind az `1..n` tartományban vannak és X[i]=j ⇔ ha Y[j]=i.
Azaz: `Xs` egy egyértelmű leképezés az `1..n` halmazon és `Ys` az inverze.

`circuit(Xs)`
`Xs` *n* hosszúságú lista. Igaz ha minden `X[i]` az `1..n` tartományba esik és `X[1], X[X[1]], ...` (n-szer ismételve) az `1..n` egy permutációja.
Azaz: Xs egy gráf Hamilton-körét adja meg. A gráf pontjai az `1,...n` számokkal vannak címkézve, `Xs[i]=j` jelentése az `i.` pontból a `j.`-be vezet a vonal.

`circuit(Xs, Ys)`
Ekvivalens a következővel: `circuit(Xs), assignment(Xs, Ys)`.

`cumulative(Starts, Durations, Resources, Limit)`
Az első három argumentum változókból/egészekből álló egyforma hossszú lista, az utolso egy változó/egész. Jelentése: a `Starts` kezdőidőpontokban elkezdett, `Durations` ideig tartó és `Resources` erőforrásigényű feladatok bármely időpontban összesített erőforrásigénye nem haladja meg a `Limit` határt. Jelöljük:

a = `min(`*S1,...,Sn*`)`, (kezdőidőpont)
b = `max(`*S1+D1,...,Sn+Dn*`)`, (végidőpont)
Rij = *Rj*, ha *Sj* `=<` *i* `<` *Sj+Dj*, egyébként *Rij* = `0`; (a j. feladat erőforrásigénye az i. időpontban)

Ezekkel a jelölésekkel a constraint jelentése:

Ri1`+...+`*Rin* `=<` *Limit*, minden *a* `=<` *i* `<` *b* esetén.

`serialized(Starts, Durations)`
A `cumulative` speciális esete, ahol az összes erőforrás-igény és a korlát is 1.

Címkéző eljárások

`indomain(X)` — korábban ismertettük.

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

```
| ?- domain([X,P], 2, 4000), P*X#=7373, minimize(labeling([], [P,X]), P).
```

`labeling(Opciók, Változók)`:
A `Változók` lista minden elemét behelyettesíti, az `Opciók` lista által előírt módon. Az alábbi négy csoport mindegyikéből legfeljebb egy opció szerepelhet.

- Változó kiválasztása:
  - `leftmost` — a legelső változó (alapértelmezés);
  - `min` ill `max` — a legkisebb alsó határú ill. a legnagyobb felső határú;
  - `ff` — first-fail elv: a legkisebb tartományú változók közül a legbaloldalibb;
  - `ffc` — a legkisebb tartományúak közül a legtöbb constraintben előforduló (ha több van, ezek közül a legbaloldalibb).
- A tartomány felbontása a kiválasztott `X` változóra:
  - `step` — `X #= B` és `X #\= B` közötti választás, ahol `B` az `X` tartományának alsó vagy felső határa (alapértelmezés);
  - `enum` — többszörös választás `X` lehetséges értekei közül;
  - `bisect` — `X #< M` és `X #>= M` közötti választás, ahol `M` az `X` tartományának középső eleme.
- A tartomány bejárási iránya:
  - `up` — alulról felfelé (alapértelmezés);
  - `down` — felülről lefelé.
- A keresett megoldások:
  - `all` — visszalépéssel az összes megoldást felsorolja; (alapértelmezés)
  - `minimize(X)` ill. `maximize(X)` — csak egy megoldást keres, `X`-re vonatkozóan a minimálisat ill. a maximálisat.

Összetettebb constraint feladatok

Egy ütemezési alapfeladat

- a rendelkezésre álló erőforrások száma: 13
- az egyes tevékenységek időtartama és erőforrásigénye:

Tevékenység	t1	t2	t3	t4	t5	t6	t7
Időtartam	16	6	13	7	5	18	4
Erőforrásigény	2	9	3	7	10	1	11

A program szövege:

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

schedule(Ss, End) :-
        length(Ss, 7),
        Ds = [16, 6,13, 7, 5,18, 4],
        Rs = [ 2, 9, 3, 7,10, 1,11],
        domain(Ss, 1, 30),
        domain([End], 1, 50),
        after(Ss, Ds, End),
        cumulative(Ss, Ds, Rs, 13),
        labeling([ff,minimize(End)], [End|Ss]).

after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).

%% End of file

| ?- schedule(Ss, End).

Ss = [1,17,10,10,5,5,1],
End = 23 ?
```


Utazó ügynök — kombinatorikus constraintek

```
:- module(tsp, [tsp/3]).
:- use_module(library(clpfd)).
:- use_module(library(lists), [append/3]).

tsp(Lab, Successor, Cost) :-
        Successor = [X1,X2,X3,X4,X5,X6,X7],
        Costs = [C1,C2,C3,C4,C5,C6,C7],
        element(X1, [0,205,677,581,461,878,345], C1),
        element(X2, [205,0,882,427,390,1105,540], C2),
        element(X3, [677,882,0,619,316,201,470], C3),
        element(X4, [581,427,619,0,412,592,570], C4),
        element(X5, [461,390,316,412,0,517,190], C5),
        element(X6, [878,1105,201,592,517,0,691], C6),
        element(X7, [345,540,470,570,190,691,0], C7),
        sum(Costs, #=, Cost),
        Predecessor = [Y1,Y2,Y3,Y4,Y5,Y6,Y7],
        Costs2 = [D1,D2,D3,D4,D5,D6,D7],
        element(Y1, [0,205,677,581,461,878,345], D1),
        element(Y2, [205,0,882,427,390,1105,540], D2),
        element(Y3, [677,882,0,619,316,201,470], D3),
        element(Y4, [581,427,619,0,412,592,570], D4),
        element(Y5, [461,390,316,412,0,517,190], D5),
        element(Y6, [878,1105,201,592,517,0,691], D6),
        element(Y7, [345,540,470,570,190,691,0], D7),
        sum(Costs2, #=, Cost),
        circuit(Successor, Predecessor),
        append(Successor, Predecessor, All),
        labeling([minimize(Cost)|Lab], All).


| ?- tsp([ff], Succs, Cost).

        Cost = 2276, Succs = [2,4,5,6,7,3,1] ?
```

Egyéb eljárások

Statisztika, válaszok

- `fd_statistics(Kulcs, Érték)`: A `Kulcs`-hoz tartozó számláló `Érték`-ét kiadja és lenullázza. Lehetséges kulcsok és számlált események: `constraints` — constraint létrehozása; `tells` — közvetlen constraint felébresztése; `asks` — reifikált constraint felébresztése; `tell_entailments` — közvetlen constraint levezethetővé válása; `ask_entailments` — reifikált constraint levezethetővé válása; `prunings` — tartomány szűkítés; `backtracks` — a tár ellentmondássá válása.
- `fd_statistics`: az összes számláló állását kiírja és lenullázza őket.
- `clpfd:full_answer`: ez egy dinamikus kampó eljárás. Ha nem sikerül, akkor a rendszer egy kérdésre való válaszoláskor csak a kérdésben előforduló változókat írja ki, és a még alvó constrainteket nem írja ki. Ha sikerül, kiírja az összes változót és a le nem vezetett constrainteket.

Felhasználói constraintek

Mit kell meghatározni egy új constraint definiálásakor?

- milyen tár-szűkítést végezzen el?
- mikor ébredjen fel újabb szűkítés megkisérlése érdekében?
- mikor fejezheti be a működését (mikor válik levezethetővé)?
- ha reifikálni is akarjuk:
  - hogyan kell végrehajtani a negáltját?
  - hogyan döntsük el a tárból való levezethetőségét?
  - hogyan döntsük el a negáltjának a levezethetőségét?

Példa: `'x+y=t'(X, Y, T)`, intervallum konzisztenciával

- `T` tartományából hagyja ki a `min(X)+min(Y)`-nél kisebb és a `max(X)+max(Y)`-nál nagyobb elemeket; ezt ismételje, valahányszor `X` vagy `Y` alsó vagy felső határa változik, mindaddig, míg `X` és `Y` értéket nem kap.
- `X`-ből hagyja ki a `min(T)-max(Y)`-nél kisebb és a `max(T)-min(Y)`-nál nagyobb elemeket; ezt ismételje ha `T` vagy `Y` határa változik.
- hasonlóan szűkítse `Y`-t is.

Constraint definiálási lehetőségek SICStusban

- globális constraintek: Prologban kell megadni a fentieket (reifikálás nincs)
- FD predikátumok: ún indexikálisok segítségével deklaratív módon lehet constrainteket definiálni, reifikáltakat is.

`'x+y=t'/3` definiálása FD predikátummal

```
'x+y=t'(X,Y,T) +:
        X in min(T) - max(Y)..max(T) - min(Y),
        Y in min(T) - max(X)..max(T) - min(X),
        T in min(X) + min(Y)..max(X) + max(Y) .
```


FD predikátumok

Egy FD predikátum 1 vagy 4 klózból áll

- A `+:` nyakjelű kötelező, a `-:`, `+?`, `-?` nyakjelűek csak reifikálandó constraintek esetén szükségesek.
- A `+:` `-:` ún szűkítő (mondó, *tell*) indexikálisok az adott constraint és negáltja tárra gyakorolt hatását írják le.
- A `+?` `-?` ún kérdező (*ask*) indexikálisok az adott constraint és negáltja levezethetőségi feltételeit definiálják.
- Az FD predikátum fejében az argumentumok különböző változók; törzsében csak ezeket a változókat tartalmazó indexikálisok lehetnek.

Indexikálisok

- Alakjuk: *Változó* `in` *Tartománykifejezés*
- *Tartománykifejezés* (eredetileg *Range*) egy halmazértékű függvény, amely a benne szereplő változóktól függoen egy tartományt generál.
- Baloldalán egy a fejben felsorolt változó áll, jobboldalán az **összes** többi változót tartalmazó tartománykifejezés.
- **Alapszabály**: Ha `X in R` szerepel egy `C` constraint definiciójában, akkor ha az `R`-beli változóknak konkrét értéket adunk, akkor az `R` tartománykifejezés értéke **pontosan** a `C` constraintet kielégítő `X` értékek halmaza kell legyen.

Példa: `'x+y=t'/3` tartomány-konzisztenciával

```
'x+y=t'(X,Y,T) +:
      X in dom(T) - dom(Y),       % {t − y|t ∈ dom(T), y ∈ dom(Y)}
      Y in dom(T) - dom(X),
      T in dom(X) + dom(Y).
```


Indexikálisok szintaxisa

```
Indexical --> X in Range

X --> variable                      { constraint-változó }

Constant --> integer
  |   inf   |   sup                 { minusz/plusz végtelen }
  |   X                             { csak behelyettesítve
                                      értékelhető ki }
Term --> Constant
  |   card(X)                       { X tartományának mérete }
  |   min(X)                        { X alsó határa }
  |   max(X)                        { X felső határa }
  |   Term + Term | Term - Term
  |   Term * Term | Term mod Term
  |   - Term
  |   Term /> Term                  { felfelé kerekített osztás }
  |   Term /< Term                  { lefelé kerekített osztás }

ConstantSet -->  {Term,...,Term}

Range --> ConstantSet
  |   dom(X)                          { X tartománya }
  |   Term..Term                      { intervallum }
  |   Range/\Range                    { metszet }
  |   Range\/Range                    { únió }
  |   \Range                          { komplementer halmaz }
  |   - Range                         { pontonkénti negáció }
  |   Range + Range | Range + Term     { pontonkénti össszeadás}
  |   Range - Range | Range - Term
  |   Term - Range                    { pontonkénti kivonás }
  |   Range mod Range | Range mod Term { pontonkénti modulo }
  |   Range ? Range                   { feltételes kifejezés }
  |   unionof(X,Range,Range)          { únió-kifejezés }
  |   switch(Term,MapList)            { kapcsoló-kifejezés }
```

Szűkítő indexikálisok kezelése

Tartománykifejezések monotonitása

- Jelöljük $D(X,S)$-sel az X változó tartományát az S tárban, $S(R)$-rel egy R tartománykifejezés értékét az S tárban.
- Egy S tár kiterjesztése S', ha minden X változóra $D(X,S') \subseteq D(X,S)$.
- Egy R tartománykifejezés egy S tárra nézve monoton, ha S minden S' kiterjesztésére $S'(R) \subseteq S(R)$.
- R S-re nézve antimonoton, ha S minden S' kiterjesztésére $S'(R) \supseteq S(R)$.
- R S-ben konstans, ha monoton és antimonoton (nem változik).
- egy indexikális (anti)monoton, ha a benne levő tartománykifejezés (anti)monoton.
- **Tétel**: ha egy `X in R` indexikális monoton egy S tárban, akkor az `X` értéktartománya S(`R`)-rel szűkíthető (vö. alapszabály).

Szűkítő indexikálisok feldolgozása

- Az `X in R` indexikálist felfüggesztjük amig kiértékelhető és monoton nem lesz (a megfelelő változók be nem helyettesítődnek).
- Az `R`-beli többi változóra megállapítjuk az ébresztési feltételeket
  - `dom(Y), card(Y)` környezetben — bármilyen változáskor
  - `min(Y)` környezetben – alsó határ változásakor
  - `max(Y)` környezetben– felső határ változásakor
- Ha az indexikális felébred:
  - Ha $D(X,S)$ és $S(R)$ diszjunktak, akkor visszalépünk.
  - a tárat az X `in` $S(R)$ constrainttel **szűkitjük** (erősítjük), azaz $D(X,S) := D(X,S) \wedge S(R)$
  - Ha $S(R)$ konstans, az **indexikálist tartalmazó constraint** levezethetővé vált.
  - Egyébként újra elaltatjuk az indexikálist.

Példák szűkítő indexikálisokra

```
't+u=c'(T,U,C) +:                  % Tartomány-konzisztens!
       T in C - dom(U),       U in C - dom(T).

't+u>=c'(T,U,C) +:
       T in C - max(U)..sup,  U in C - max(T)..sup.

'ax+c=t'(A,X,C,Y) +:
       X in (min(Y) - C) /> A .. (max(Y) - C) /< A,
       Y in  min(X)*A + C     .. max(X)*A + C.

'p\\/q'(P, Q, B) +:
       B in (min(P)+min(Q))/>2 .. (max(P)+max(Q))/>2,
       P in min(B)*2 - max(Q) - 1  .. max(B)*2 - min(Q),
       Q in min(B)*2 - max(P) - 1  .. max(B)*2 - min(P).
% | ?- domain([P,Q,B], 0, 1), 'p\\/q'(P, Q, B), P=1.
%                           B = 1, P = 1, Q in 0..1 ?

'abs(x-y)>c'(X, Y, C) +:
       X in (inf .. max(Y)-C-1) \/ (min(Y)+C+1 .. sup),
       Y in (inf .. max(X)-C-1) \/ (min(X)+C+1 .. sup).
% | ?- 'abs(x-y)>c'(X,Y,3), X in 3..5.
%                           X in 3..5, Y in(inf..1)\/(7..sup) ?

no_threat_3(X, Y, I) +:
       X in \({Y} \/ {Y+I} \/ {Y-I}),
       Y in \({X} \/ {X+I} \/ {X-I}).

% Hibás constraint, sérti az alapszabályt!
'x=<y=<z'(X, Y, Z) +:
       Y in min(X)..max(Z),
       Z in min(Y).. sup,
       X in inf..max(Y).
% | ?- 'x=<y=<z'(15, 5, Z).
%                           Z in 5..sup ?
```


Bonyolultabb tartománykifejezések

Únió-kifejezés

`unionof(X, H, T)` — ahol `X` változó, `H` és `T` tartománykifejezések. Kiértékelése egy S tárban: kiértékeljük `H`-t, legyen ennek értéke $\{x_1, \ldots, x_n\}$. Képezzük a T_i kifejezéseket úgy, hogy `T`-ben `X` helyébe x_i-t írjuk. Ekkor az únió-kifejezés értéke a $S(T_1), \ldots, S(T_n)$ halmazok uniója. Ha $S(H)$ végtelen, felfüggesztjük. Példa:

```
% Maximálisan szűkítő, de nagyon hatékonytalan!
no_threat_4(X, Y, I) +:
       X in unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
       Y in unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
```

Feltételes kifejezés

`Felt ?  Tart` – ahol `Felt` és `Tart` tartománykifejezések. Ha S(`Felt`) üres halmaz, akkor a feltételes kifejezés értéke is üres halmaz, egyébként pedig azonos S(`Tart`) értékével. Példa:

```
'x=<y=<z'(X, Y, Z) +:       % Ez már helyes!
       Y in min(X)..max(Z),
       Z in ((inf..max(Y)) /\ dom(X)) ? (min(Y).. sup),
       X in ((min(Y)..sup) /\ dom(Z)) ? (inf..max(Y)).
```

Használható a kiértékelés késleltetésére: `( Felt?(inf..sup) \/ Tart )` eredménye S(`Tart`), ha S(`Felt`) üres, egyébként `inf..sup`. Az ilyen szerkezetekben `Tart` értékét ki sem számoljuk, amíg `Felt` nem üres. Példa:

```
% Maximálisan szűkít, kicsit kevésbé hatékonytalan
no_threat_5(X, Y, I) +:
    X in (4..card(Y)) ? (inf..sup) \/
          unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
    Y in (4..card(X)) ? (inf..sup) \/
          unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
```


Kapcsoló-kifejezés

`switch(T, MapList)` — ahol `T` a `term` szintaxisának megfelelő kifejezés, `MapList` pedig `integer-Range` alakú párokból álló lista. Ha S(`T`) értéke Key és `MapList` tartalmaz egy $Key - Expr$ párt, akkor a kapcsoló értéke $S(Expr)$, egyébként üres. Például az alábbi két definició hatása azonos:

```
p(X, Y) :- relation(X, [1-{1},2-{1,2},3-{1,2,3}], Y).

q(X, Y) +:
  X in unionof(B,dom(Y),switch(B,[1-{1,2,3},2-{2,3},3-{3}])),
  Y in unionof(B,dom(X),switch(B,[1-{1},2-{1,2},3-{1,2,3}])).
```

Reifikálható constraintek definiálása

Négy klózból álló FD-predikátum szükséges, példa:

```
'x\\=y'(X,Y) +:        % 1. a constraintet szükitő indexikálisok
       X in \{Y},
       Y in \{X}.
'x\\=y'(X,Y) -:        % 2. a negáltját szükitő indexikálisok
       X in dom(Y),
       Y in dom(X).
'x\\=y'(X,Y) +?        % 3. a levezethetőséget kérdező
       X in \dom(Y).   %                              indexikális
'x\\=y'(X,Y) -?        % 4. a negált levezethetőségét kérdező
       X in {Y}.       %                              indexikális
```

A kérdező klózok csak egyetlen indexikálist tartalmazhatnak.

`'x\\=y'(X,Y) #<=> B` végrehajtásakor

- A 3. klóz figyeli, hogy `X` és `Y` tartománya diszjunkttá vált-e, ha igen, `B=1`;
- A 4. klóz figyeli, hogy `X=Y` igaz-e, ha igen `B=0`;
- egy külön démon figyeli, hogy `B` behelyettesítődött-e, ha igen, és `B=1`, meghivja az 1. klózt, ha `B=0`, akkor a 2. klózt.

Kérdező indexikálisok feldolgozása

- Az `X in R` indexikálist felfüggesztjük amig kiértékelhető és antimonoton nem lesz (a megfelelő változók be nem helyettesítődnek).
- Az ébresztési feltételek (`Y` az `R`-ben előforduló változó):
  - `X` tartományának bármilyen változáskor
  - `dom(Y), card(Y)` környezetben — bármilyen változáskor
  - `min(Y)` környezetben – alsó határ változásakor
  - `max(Y)` környezetben – felső határ változásakor
- Ha az indexikális felébred:
  - Ha $D(X,S) \subseteq S(R)$ akkor a constraint levezethetővé vált.
  - Egyébként, ha $D(X,S)$ és $S(R)$ diszjunktak, valamint $S(R)$ konstans, akkor a constraint negáltja levezethetővé vált.
  - Egyébként újra elaltatjuk az indexikálist.

Példa

```
'x=<y'(X,Y) +:
        X in inf..max(Y),
        Y in min(X)..sup.
'x=<y'(X,Y) -:
        X in (min(Y)+1)..sup,
        Y in inf..(max(X)-1).
'x=<y'(X,Y) +?
        X in inf..min(Y).
'x=<y'(X,Y) -?
        X in (max(Y)+1)..sup.
```


FD-predikátumok, indexikálisok összefoglalása

- Legyen `C(Y1, ...)` egy FD-predikátum, amelyben szerepel egy

  `Yi in R(Y1, ...)`

  indexikális. A baloldali `Yi` változót néha `X`-nek nevezzük. Az `R` tartománykifejezéshez tartozó halmazfüggvényt nevezzük R-nek.
- (definició:) Az indexikális az alábbi relációt definiálja:

  C = { $< y_1, \ldots >$ | $y_k \in$ `inf..sup` $\wedge y_i \in R(\{y_1\}, \ldots)$}
- (alapszabály:) Egy FD-predikátum csak akkor értelmes, ha a pozitív klózaiban levő összes indexikális ugyanazt a relációt definiálja; továbbá a negatív klózaiban levő összes indexikális ennek a relációnak a negáltját (komplemensét) definiálja.
- Ha R monoton egy S tárra nézve, akkor $S(R)$-ről belátható, hogy minden olyan x értéket tartalmaz, amelyek (az S által megengedett y_j értékekkel együtt) a C relációt kielégítik. Ezért szűkítő indexikálisok esetén jogos az X tartományát $S(R)$-rel szűkíteni.
- Ha R antimonoton egy S tárra nézve, akkor $S(R)$-ről belátható, hogy minden olyan x értéket kizár, amelyekre (az S által megengedett y_j értékekkel együtt) a C reláció nem áll fenn. Ezért kérdező indexikálisok esetén, ha $D(X,S) \subseteq S(R)$, jogos a constraintet az S tárbol levezethetőnek tekinteni.
- A fentiek miatt természetesen adódik az indexikálisok felfüggesztési szabálya: a szűkítő indexikálisok végrehajtását mindaddig felfüggesztjük, amig monotonná nem válnak; a kérdező indexikálisok végrehajtását mindaddig felfüggesztjük, amig antimonotonná nem válnak.
- **Az indexikálisok deklaratív volta:** Ha a fenti alapszabályt betartjuk, akkor a `clpfd` megvalósítás az FD-predikátumot helyesen valósítja meg, azaz mire a változók teljesen behelyettesítetté válnak az FD-predikátum akkor és csak akkor for sikeresen lefutni, vagy 1-gyé tükroződni (reifikálódni), ha a változók értékei a predikátum által definiált relációhoz tartoznak. Az indexikális megfogalmazásán csak az múlik, hogy a nem-konstans tárak esetén milyen jó lesz a szűkítő ill. kérdező viselkedése.

Constraintek automatikus fordítása indexikálisokká

Indexikálissá fordítandó constraint

- Formája: *Head* `+:` *Constraint*.
- Ebben *Constraint* lehet
  - aritmetikai (csak lineáris kifejezésekre);
  - `relation/3` és `element/3` szimbolikus constraint.
- Csak a `+:` nyakjel használható, ezek a constraintek nem reifikálhatóak.

Mi a különbség?

- `p(X,Y,U,V) :- X+Y #< U+V).` törzse `clpfd` könyvtári hívásokra vagy a `scalar_product` globális korlátra fordul.
- `p(X,Y,U,V) +: X+Y #< U+V).` egyetlen FD predikátummá válik: `p(X,Y,U,V) +: X in` R`(Y,U,V), Y in ..., U in ..., V in ....`
- Jelenleg az első változat kevesebb helyet foglal el és gyorsabb is. (SICStus 3.5-ben nem volt még `scalar_product`, akkor még megérte az aritmetikai indexikálist alkalmazni, de most már nem.)
- A `relation/3` és `element/3` szimbólikus constraintek unió- és kapcsoló-kifejezésekké fordulnak (lineáris méret).
- Mivel ezek végrehajtási ideje függ a tartomány méretétől, és az első alkalmazás nem különbözik a többitől, ezért vigyázni kell a kezdő-tartományok megfelelő beállítására.
- A torpedó feladatban a `relation` korlátok formájának hatása:

Probléma/forma	:-	+:
fules2	12.31	10.67
dense-clean	4.02	2.77
dense-collapse	1.79	1.29

Globális constraintek

Definiálás módja

- `fd_global(Constraint, State, Susp)`: `Constraint` végrehajtásának elindítása, `State` kezdőállapottal, `Susp` ébresztési listával (elemei: `dom(X)`, `min(X)`, `max(X)`, `minmax(X)`, `val(X)`).
- `clpfd:dispatch_global(Constraint, State0, State, Actions)`: A kampó-eljárás törzse definiálja a `Constraint` felébredésekor elvégzendőket: `State0` a régi, `State` az új állapot. Az `Actions` lista elemei lehetnek: `exit`, `fail`, `X=V`, `X in R`, `X in_set S`, `call(Module:Goal)`.

Tartományok lekérdezése

- `fd_min(X, Min)`; `fd_max(X, Max)`; `fd_size(X, Size)`: Az `X` változó tartományának pillanatnyi alsó határa `Min`, felső határa `Max`, mérete `Size`. Ha valamelyik nem korlátos, az eljárás meghiúsul.
- `fd_set`: lásd alább

Segédeszköz: FD-halmazok

- `fd_set(X, Set)`: `X` tartománya a `Set` FD-halmaz.
- `X in_set Set`: `X`-et `Set`-re szűkíti.
- Az FD-halmaz: diszjunkt intervallumok rendezett listája. Mint absztrakt adattípus, alapműveletei:
  - `empty_fdset(S)`: `S` az üres FD-halmaz.
  - `fdset_parts(S, Min, Max, Rest)`: `S` első intervalluma `Min..Max`, a maradék FD-halmaz: `Rest`. FD-halmaz építésére is jó.
  - `is_fdset(S)`: `S` egy korrekt FD-halmaz.
- További, kényelmi szolgáltatások, pl.
  - `fdset_singleton(Set, Elt)`: `Set` az egyetlen `Elt`-ből áll.
  - `fdset_union/3`, `fdset_intersection/3`, `fdset_complement/2`.
  - `fdset_member(Elt, Set)`: `Elt` eleme a `Set` FD-halmaznak.

Példa: exactly/3 (korábbi pontosan/3)

```
exactly(I, Xs, N) :-
    dom_susps(Xs, Susp), fd_global(exactly(I,Xs,N), Xs/N, Susp).

dom_susps([], []).
dom_susps([X|Xs], [dom(X)|Susp]) :- dom_susps(Xs, Susp).

:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_), Xs0/N0, Xs/N, Actions) :-
    ex_filter(Xs0, Xs, N0, N, I), length(Xs, M),
    (   N=:=0 -> Actions = [exit|Ps], ex_neq(Xs, I, Ps)
    ;   N=:=M -> Actions = [exit|Ps], ex_eq(Xs, I, Ps)
    ;   N>0, N<M -> Actions = []
    ;   Actions = [fail]
    ).

ex_filter([], [], N, N, _).
ex_filter([X|Xs], Ys, L, N, I) :-
    X==I, !, M is L-1, ex_filter(Xs, Ys, M, N, I).
ex_filter([X|Xs], Ys0, L, N, I) :-
    fd_set(X, Set), fdset_member(I, Set), !,
    Ys0 = [X|Ys], ex_filter(Xs, Ys, L, N, I).
ex_filter([_|Xs], Ys, L, N, I) :- ex_filter(Xs, Ys, L, N, I).

ex_neq(Xs, I, Ps) :- fdset_singleton(Set0, I),
    fdset_complement(Set0, Set), eq_all(Xs, Set, Ps).

ex_eq(Xs, I, Ps) :- fdset_singleton(Set, I), eq_all(Xs, Set, Ps).

eq_all([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :- eq_all(Xs, Set, Ps).

| ?- exactly(5,[A,B,C],1), A=5.
     A = 5, B in(inf..4)\/(6..sup), C in(inf..4)\/(6..sup)
| ?- exactly(5,[A,B,C],1), A in 1..2, B in 3..4.
     C = 5, A in 1..2, B in 3..4
```


Példa: exactly/3 továbbfejlesztése

`exactly2(I, Xs, N)`

- elfogad korlát-változót az `N` (számláló) argumentumban
- állapot: `Xs1/D`, ahol: `Xs1` a lista fennmaradó része, `D =< 0`, a listából kihagyott `I` értékű változók számának negáltja.

```
exactly2(I, Xs, N) :-
        dom_susps(Xs, Susp),
        length(Xs, Len), N in 0..Len,
        fd_global(exactly2(I,Xs,N), Xs/0, [minmax(N)|Susp]).

:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly2(I,_,N), Xs0/D0, Xs/D, Actions) :-
        exactly_solver(I, Xs0, Xs, N, D0, D, Actions).

exactly_solver(I, Xs0, Xs, N, D0, D, Actions) :-
        ex_filter(Xs0, Xs, D0, D, I),
        length(Xs, M), NewMin is -D, NewMax is NewMin+M,
        fd_min(N, MinN), fd_max(N, MaxN),
        (   MaxN=:=NewMin ->
            Actions = [exit,N=MaxN|Ps], ex_neq(Xs, I, Ps)
        ;   MinN=:=NewMax ->
            Actions = [exit,N=MinN|Ps], ex_eq(Xs, I, Ps)
        ;   Actions = [N in NewMin..NewMax]
        ).

| ?- _L=[A,B,C,D], domain(_L, 1, 4),
     A #< B, B #=< C, C#<D, exactly2(4, _L, N).

A in 1..2, B in 2..3, C in 2..3, D in 3..4,
N in 0..1 ?
```


Probléma az exactly2 korláttal

```
| ?- L = [N,1], N in {0,2}, exactly2(0, L, N).

L = [0,1],
N = 0 ? ;
no
| ?- L = [N,1], N in {0,2}, exactly2(0, L, N).
 +      1      1 Call: exactly_solver(0,[_645,1],_781,
                       _645,0,_782,_768) ? @
| :- exactly_solver(_,_,_,N,_,_,_)^(fd_set(N, Set),
     write(Set), nl).
[[0|0],[2|2]]
 +      1      1 Call: exactly_solver(0,[_645,1],_781,
                       _645,0,_782,_768) ? s
 +      1      1 Exit: exactly_solver(0,[_645,1],[_645],
                       _645,0,0,[_645 in 0..1]) ?
L = [0,1],
N = 0 ? ;

no
```

Egy lehetséges javítás

```
exactly3(I, Xs, N0) :-
        dom_susps(Xs, Susp),
        length(Xs, Len), N0 in 0..Len, N #= N0,
        fd_global(exactly2(I,Xs,N), Xs/0, [minmax(N)|Susp]).

| ?- L = [N,1], N in {0,2}, exactly3(0, L, N).
no
```


Esettanulmány: Négyzetdarabolás

A feladat

- Adott egy nagy négyzet oldalhosszúsága, pl: 10.
- Adottak kis négyzetek oldalhosszúságai, pl `[6,4,4,4,2,2,2,2]` (területösszegük megegyezik a nagy négyzet területével).
- A kis négyzetekkel pontosan le kell fedni a nagyot (meghatározandók a kis négyzetek koordinátái), pl:
  `Xs = [1,7,7,1,5,5,7,9]`
  `Ys = [1,1,5,7,7,9,9,9]`
  `Ss = [6,4,4,4,2,2,2,2]`

Próba-adatok

- `10, [6,4,4,4,2,2,2,2]`
- `20, [9,8,8,7,5,4,4,4,4,4,3,3,3,2,2,1,1]`
- `112, [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11, 9,8,7,6,4,2]`
- `175, [81,64,56,55,51,43,39,38,35,33,31,30,29,20,18,16, 14,9,8,5,4,3,2,1]`

Négyzetdarabolás: Prolog megoldás

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

squares_prolog(Ss, Size, SXYs) :-
        MSize is -Size,
        squares(Ss, [MSize/Size,Size/0], Size, SXYs).

squares([], _, _, [])  :- !.
squares(Ss0, L0, Size, SXYs) :-
        L0=[V0/_|_], Y0 is Size+1+V0,
        hole(L0, 1-Y0, XY, L1, L1T, L2),
        fill_hole(L2, L1T, XY, Size, Ss0, Ss, SXYs1, SXYs),
        squares(Ss, L1, Size, SXYs1).

fill_hole([V0/0,V/H|L], [V1/H|L], _XY, _Size,
                        Ss0, Ss0, SXYs0, SXYs0) :-
        !, V1 is  V0+V.
fill_hole([V/H|L0], [V1/S|L1], X-Y, Size,  Ss0, Ss,
                        SXYs0, [s(S,X,Y)|SXYs]) :-
        MaxS is min(H, Size+1-Y),
        select(S, Ss0, Ss1), S =< MaxS,
        V1 is V+S, MS is -S, H1 is H-S, X1 is X+S,
        fill_hole([MS/H1|L0], L1, X1-Y, Size, Ss1, Ss, SXYs0, SXYs).

hole([V1/0,V2/H|L], XY0, XY, L1, L1T, L2) :-
        !, V is V1+V2,
        hole([V/H|L], XY0, XY, L1, L1T, L2).
hole([V/H1,0/H2|L], XY0, XY, L1, L1T, L2) :-
        !, H is H1+H2,
        hole([V/H|L], XY0, XY, L1, L1T, L2).
hole([V1/H1|L], XY, XY, L1, L1, [V1/H1|L]) :-
        H1 > 0, V1 < 0, L = [V2/_|_], V2>0, !.
hole([VH1,VH2|L], X0-Y0, XY, [VH1|L1], L1T, L2) :-
        VH1=_/H1, VH2=V2/_, X1 is X0+H1, Y1 is Y0+V2,
        hole([VH2|L], X1-Y1, XY, L1, L1T, L2).
```


Négyzetdarabolás: naív `clpfd` megoldás

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

squares_spec(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_asymmetry(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        labeling([], Xs), labeling([], Ys).

generate_coordinates([], [], [], _).
generate_coordinates([X|Xs], [Y|Ys], [S|Ss], Size) :-
        Sd is Size-S+1, domain([X,Y], 1, Sd),
        generate_coordinates(Xs, Ys, Ss, Size).

state_asymmetry([X|_], [Y|_], [D|_], Limit) :-
        UB is (Limit-D+2)>>1, X in 1..UB, Y #=< X.

state_no_overlap([], [], []).
state_no_overlap([X|Xs], [Y|Ys], [S|Ss]) :-
        state_no_overlap(X, Y, S, Xs, Ys, Ss),
        state_no_overlap(Xs, Ys, Ss).

state_no_overlap(_, _, _, [], [], []).
state_no_overlap(X, Y, S, [X1|Xs], [Y1|Ys], [S1|Ss]) :-
        no_overlap_spec(X, Y, S, X1, Y1, S1),
        state_no_overlap(X, Y, S, Xs, Ys, Ss).

no_overlap_spec(X1, _Y1, S1, X2, _Y2, _S2) :-
        X1+S1 #=< X2.
no_overlap_spec(X1, _Y1, _S1, X2, _Y2, S2) :-
        X2+S2 #=< X1.
no_overlap_spec(_X1, Y1, S1, _X2, Y2, _S2) :-
        Y1+S1 #=< Y2.
no_overlap_spec(_X1, Y1, _S1, _X2, Y2, S2) :-
        Y2+S2 #=< Y1.
```


Négyzetdarabolás: diszjunktív constraintek

A diszjunkció lehetséges megvalósításai

- spekulatív: Prolog választási pontokkal (lásd `squares_spec`).
- reifikáció(számosság)-alapú: a `#\/` müvelettel, pl:
  `X1+5 #=< X2 #\/ X2+5 #=< X1`.
- (gyenge) konstruktív diszjunkció:
  `domain([X1,X2], 0, 6), X1+5 #=< X2 \/ X2+5 #=< X1`
  `X1 in (0..1)\/(5..6), X2 in (0..1)\/(5..6)`
- teljes konstruktív diszjunkció:
  `domain([X,Y], 0, 20), X+Y #= 20, X #=<5 \/ Y #=< 5.`
  `X in (0..5)\/(15..20), Y in (0..5)\/(15..20)`

Számosság-alapú `no_overlap`

```
no_overlap_card(X1, Y1, S1, X2, Y2, S2) :-
        X1+S1 #=< X2 #<=> B1,
        X2+S2 #=< X1 #<=> B2,
        Y1+S1 #=< Y2 #<=> B3,
        Y2+S2 #=< Y1 #<=> B4,
        B1+B2+B3+B4 #>= 1.
```

Gyenge konstruktív diszjunkción (wcd) alapuló `no_overlap`

```
no_overlap_wcd1(X1, Y1, S1, X2, Y2, S2) :-
  call( abs(2*(X1-X2)+(S1-S2)) #>= S1+S2 #\/
        abs(2*(Y1-Y2)+(S1-S2)) #>= S1+S2 ).
```

```
no_overlap_wcd2(X1, Y1, S1, X2, Y2, S2) +:
  X1 in ((((min(Y1)+S1)..max(Y2))\/((min(Y2)+S2)..max(Y1))) ? (inf..sup))
        \/ ((inf..max(X2)-S1) \/ (min(X2)+S2..sup)),
  X2 in ((((min(Y1)+S1)..max(Y2))\/((min(Y2)+S2)..max(Y1))) ? (inf..sup))
        \/ ((inf..max(X1)-S2) \/ (min(X1)+S1..sup)),
  Y1 in ((((min(X1)+S1)..max(X2))\/((min(X2)+S2)..max(X1))) ? (inf..sup))
        \/ ((inf..max(Y2)-S1) \/ (min(Y2)+S2..sup)),
  Y2 in ((((min(X1)+S1)..max(X2))\/((min(X2)+S2)..max(X1))) ? (inf..sup))
        \/ ((inf..max(Y1)-S2) \/ (min(Y1)+S1..sup)).
```


Négyzetdarabolás: redundáns constraintek 1.

Kapacitás-constraintek

- minden sorban és minden oszlopban vizsgáljuk meg a metszett négyzeteket,
- kössük ki, hogy ezek oldalhossz-összege a nagy négyzet oldalával megegyezik.

A program

```
squares_cap(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_asymmetry(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(Xs, Sizes, Size),
        state_capacity(Ys, Sizes, Size),
        labeling([], Xs), labeling([], Ys).

state_capacity(Cs, Sizes, Limit) :-
        state_capacity(1, Limit, Cs, Sizes).

state_capacity(Pos, Limit, _, _) :-
        Pos>Limit, !.
state_capacity(Pos, Limit, Cs, Sizes) :-
        accumulate(Cs, Sizes, Pos, Bs),
        scalar_product(Sizes, Bs, #=, Limit),
        Pos1 is Pos+1,
        state_capacity(Pos1, Limit, Cs, Sizes).

accumulate([], [], _, []).
accumulate([C|Cs], [S|Ss], Pos, [B|Bs]) :-
        Crutch is Pos-S+1,
        C in Crutch .. Pos #<=> B,
        accumulate(Cs, Ss, Pos, Bs).
```

Négyzetdarabolás: redundáns constraintek 2.

Kumulativ-constraint

- A négyzetdarabolás tekinthető ütemezési problémának is
- Alkalmazzuk a `cumulative` constraint-et mindkét tengely irányában

A program

```
squares_cum(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_asymmetry(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        cumulative(Xs, Sizes, Sizes, Size),
        cumulative(Ys, Sizes, Sizes, Size),
        labeling([], Xs), labeling([], Ys).
```


Négyzetdarabolás: speciális címkézés

A címkézés lényege:

- keressük meg a legkisebb alsó határt,
- sorra próbáljuk a változókat a legkisebb határ-értékkel behelyettesíteni,
- ha egy változóra ez nem sikerül, zárjuk ki az alsó határt a tartományából.

A program

```
squares_dual(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(Xs, Sizes, Size),
        state_capacity(Ys, Sizes, Size),
        dual_labeling(Xs, 1, Size),
        dual_labeling(Ys, 1, Size).

dual_labeling([], _, _) :- !.
dual_labeling(L, I, Limit) :-
        dual_labeling(L, L1, I, Limit, J),
        dual_labeling(L1, J, Limit).

dual_labeling([], [], _, J, J).
dual_labeling([X|L1], L2, I, J0, J) :-
        (   integer(X) -> dual_labeling(L1, L2, I, J0, J)
        ;   X = I, dual_labeling(L1, L2, I, J0, J)
        ;   X #> I,
            fd_min(X, J1), J2 is min(J0,J1), L2 = [X|L3],
            dual_labeling(L1, L3, I, J2, J)
        ).
```


A négyzetdarabolás megoldásainak összehasonlítása

Futási idő (másodperc) — visszalépések száma

variáns/méret	10		20		112		175	
Prolog!!	0.00	0	0.039	1678	70.951	3.3M	>200	
spec	2.372	34K						
cap+spec	2.384	34K						
spec+dual	2.372	34K						
cap+spec+dual	2.386	34K						
cap+card	0.045	0	0.297	18				
cap+wcd1	0.059	0	0.353	18				
cap+wcd2	0.019	0	0.110	18				
cap+card+dual	0.045	0	0.225	0	3.244	115	5.631	105
cap+wcd1+dual	0.059	0	0.298	0	3.238	115	5.094	105
cap+wcd2+dual	0.021	0	0.095	0	2.810	115	5.286	105
cum+card+dual	0.033	0	0.168	0	1.042	139	0.854	67
cum+wcd1+dual	0.044	0	0.237	0	1.078	139	0.957	67
cum+wcd2+dual	0.009	0	0.046	0	0.609	139	0.444	67
cap+cum+card+dual	0.045	0	0.229	0	2.327	100	2.541	65
cap+cum+wcd1+dual	0.058	0	0.298	0	2.242	100	2.624	65
cap+cum+wcd2+dual	0.023	0	0.105	0	1.891	100	2.228	65

A SICStus 3.8-as változatában a `cumulative` korlát alapértelmezése gyengült, a fenti `cum` futásokban paraméterezése:

```
cumulative(Xs, Sizes, Sizes, Size, [edge_finder(true)])
```


Torpedó — 1999-es házifeladadat

A feladat

- Téglalap alakú táblázat.
- Egyenes hajókat kell elhelyezni benne (kiírás: 1, 2, 3 és 4 hosszúak, minta-megvalósítás: tetszőleges hosszúak)
- A hajók különböző színűek lehetnek.
- Minden szín esetén adott:
  - minden hajóhosszhoz: az adott színű és hosszú hajók száma;
  - minden sorra és oszlopra: az adott színű hajó-darabok száma;
  - ismert hajó-darabok a táblázat mezőiben.
- Színfüggetlenül adott: ismert torpedó-mentes (tenger) mezők

Példa — adat és eredmény egy táblában

```
      1 2 3 4 5        % oszlopsorszám
      0 1 1 1 0        % 1. szín oszlopösszegek

1  2  = * r : :  0     %
2  0  : : : : #  1     %
3  0  # : : : :  1     % 1. és 2. szin sorösszegek
4  1  # : : * :  1     %

      2 0 0 0 1        % 2. szín oszlopösszegek

% Ismert mezők, > 1 hossz:  (1. szín)     (2. szín)   (tenger)
%                              u             U
%                          l   m   r     L   M   R
%                              d             D
% Ismert mezők (1 hosszúak):   o             O          =
% Kikövetkeztetett mezők:      *             #          :
```

Torpedó — adattípusok

```
% Példa-adat:
named_problem(simple,
    descr([1-1],
         [c([1,1], [2,0,0,1], [0,1,1,1,0], [t(1,3,r)]),
          c([1,1], [0,1,1,1], [2,0,0,0,1], [])]
        )).

% :- type torp_descr --->    descr(known_empties, colours_descr).
% :- type known_empties ==   list(empty_field).
% :- type empty_field --->   row_index-column_index.
% :- type colours_descr ==   list(colour_descr).
% :- type colour_descr --->  c(list(ship_count),
%                              row_occs, column_occs,
%                              list(known_piece)).
% :- type row_occs ==        list(occ).
% :- type column_occs ==     list(occ).
% :- type known_piece --->   t(row_index, column_index, kind).
% :- type row_index ==       int. % 1,2,...
% :- type column_index ==    int. % 1,2,...
% :- type occ ==             int.
% :- type ship_count ==      int.
% :- type kind --->          u ; l ; m ; r ; d ;  o.

% Példa-eredmény:
% [[ = ,l-1,r-1, = , = ],
%  [ = , = , = , = ,o-2],
%  [u-2, = , = , = , = ],
%  [d-2, = , = ,o-1, = ]]

% :- type board  ==          list(row).
% :- type row ==             list(field).
% :- type field --->         =  /* empty */
%                          ; kind-colour.  /* piece */
% :- type colour ==          int. % 1,2,...
```


Torpedó — példafutás

```
% Done on a 200 MHz Pentium under Linux.
| ?- time_torpedo(fules).

      1 2 3 4 5 6 7 8 9 0 1 2 3
      0 2 2 2 2 2 2 1 1 1 0 4 1

1  0  : : : : : : : # : : : : : :  1
2  3  : * r : = : # : : # : * :  2
3  1  : : : : : # : # : : : : * :  2
4  0  : # : : : : : : : : : : : :  1
5  4  : # : l * * * : : # : : :  2
6  0  : M : : : : : : : # : : :  2
7  3  : # : : : : : : o : : : * *  1
8  0  : : : : : : O : : : : : : :  1
9  4  = * * * : : : : * = : : :  0
0  1  : : : : : : : : : : : : o :  0
1  3  # : : : * * * : : # : : :  2
2  0  # : : : : : : : : : = : : #  2
3  1  # : : : : : # # : : * : : #  4

      3 4 0 0 1 2 4 0 0 4 0 0 2

Solution 1:    1.690 sec, 0 backtracks.
Failure:       0.000 sec, 0 backtracks.
Solutions found: 1 (all), 1.790 sec, 0 backtracks.
```


Torpedó — modellezés

Mik legyenek a korlát-változók?

- Minden hajóhoz: irány (vízsz./függ) és a kezdőpont koordinátái — kevés változó, de szimmetria problémák, bonyolultabb korlátok, sok diszjunktív korlát;
- Minden mezőhöz: mi található ott: hajó-darab vagy tenger — sok változó, egyszerűbb korlátok; **választott megoldás**

Milyen értékkészletet adjunk a korlát-változóknak (mezőknek)?

- adott színű hajó-darab vagy tenger — egyszerű kódolás; nehéz „intelligensen” címkézni
- megkülonböztetjük a hajó-darabokat:
  - az előre kitöltött mezőknek megfelelő darabok (u,l,m,r,d,o) — diszjunktív korlátok;
  - részletesebb bontás: a mezőket megkülonböztetjük a hajó hossza, iránya, a darab hajón belüli pozíciója szerint; **választott megoldás**

Hány változóval ábrázoljunk egy mezőt?

- külön változó mutatja a szín, hossz, irány és pozíció értékét — egyszerű kódolás, a szűkítés gyenge;
- egyetlen változó mutatja az összes jellemzőt — bonyolult kódolás, hatékonyabb szűkítés; **választott megoldás**
- az utóbbi megoldás jellemzője: ha egy mező egy nem-tenger értéket kap, akkor a teljes hajó meghatározottá válik.

Torpedó mintamegoldás — változók

Korlát-változók

- Minden mezőnek egy változó felel meg.
- Az értékek kódolási elvei (`max` címkézéshez)
  - az irányított hajók orra (`l` és `u`) kapja a legmagasabb kódokat,
  - ezen belül a hosszabbak kapják a nagyobb kódokat
  - adott hossz esetén az irány és a szín sorrendje nem fontos
  - az irányított hajók nem-orr elemeinek kódolása nem lényeges (címkézéskor az orr-elemek helyettesítődnek be)
  - az egy-hosszú hajók (hajódarabok) kódja a legalacsonyabb
  - a tenger kódja minden hajónál alacsonyabb
- Példa-kódolás (1 szín, max 3 hosszú hajók, hij = vízszintes, i hosszú hajó j-edik darabja, stb.):

```
0:       tenger
1:       h11 = v11      % 1-hosszú hajó
2..4     v33  h22 h32   % nem-orr-elemek
5..7     v32 v22  h33   % nem-orr-elemek
8..9     h21 v21        % orr-elemek
10..11   h31 v31        % orr-elemek
```

A kódoláshoz kapcsolódó korlátok

- `coded_field_neighbour(Dir, CF0, CF1)`: `CF0` kódolt mező `Dir` irányú szomszédja `CF1`, ahol `Dir` lehet `horiz, vert, diag`. Például `| ?- coded_field_neighbour(horiz, 0, R). => R in \{3,4,7}`.
- `group_count(Group, CFs, Count, Env)`: a `Group` csoportba tartozó elemek száma a `CFs` listában `Count`, ahol a futási környezet `Env`. Itt `Group` például lehet `all(Clr)`: az összes `Clr` színű hajódarab.

Torpedó mintamegoldás — korlátok

Alapvető korlátok

- A bemenetben megadott mezők adott csoportra való megszorítása.
- Színenként az adott sor- és oszlopösszegek meghatározása.
- A hajóorr-darabok megszámolásával az adott hajófajta darabszámának biztosítása (az egy hosszú darabokat soronként számolva).
- A vízszintes, függőleges és átlós irányú szomszédos mezőkre vonatkozó korlátok

Redundáns korlátok

- `count_empties`: minden sorra és oszlopra a tenger-mezők számának előírása.
- `count_ships_occs`: sorösszegek alternatív kiszámolása:

$$\sum_{i<hosszak} i * \mathtt{hi1} + \sum_{1<i<hosszak, j \le i} \mathtt{v}ij \qquad (1)$$

  Analóg módon az oszlopösszegekre is.
- `count_ones_columns`: az egy hosszú darabok számát az oszloponkénti előfordulások összegekeként is meghatározzuk

Torpedó mintamegoldás — korlátok (folyt.)

Korlátok megvalósítási variánsai

- `relation(clause/indexical)`: a vízszintes és függőleges szomszédsági relációt a `relation/3` meghívásával, vagy indexikálisként való fordításával valósítjuk meg (alapértelmezés: `indexical`).
- `diag(reif/ind_arith/ind_cond)`: az átlós szomszédsági relációt reifikációs alapon, aritmetikát vagy feltételes tartománykifejezést használó indexikálisként valósítjuk meg (alapértelmezés: `ind_cond`).

Az átlós szomszédsági reláció megvalósításai

```
% CF1 is a diagonal neighbour of CF2.
diagonal_neighbour_reif(CF1, CF2) :-
        CF1 #= 0 #\/ CF2 #= 0.

diagonal_neighbour_arith(CF1, CF2) +:
        CF1 in 0 .. (1000-(min(CF2)/>1000)*1000),
        CF2 in 0 .. (1000-(min(CF1)/>1000)*1000).

diagonal_neighbour_cond(CF1, CF2) +:
        CF1 in (min(CF2)..0) ? (inf..sup) \/ {0},
        CF2 in (min(CF1)..0) ? (inf..sup) \/ {0}.
```


Torpedó mintamegoldás — címkézés

Címkézési variánsok — `label(`*`Variáns`*`)` opciók

- `plain`: `labeling([max,down], Mezők)`.
- `max_dual`: a négyzetkirakáshoz hasonlóan a legmagasabb *értékeket* próbálja a változóknak értékül adni.
- `ships`: speciális címkézés, minden hosszra, a legnagyobbtól kezdve, minden színre az adott színű és hosszú hajókat sorra elhelyezi (alapértelmezés).

Címkézés közbeni szűrés

- a konstruktív diszjunkció egy egyszerű formája
- sorra az összes mezőt megpróbáljuk „tenger"-re helyettesíteni, ha ez azonnal meghiúsulást okoz, akkor ott hajó-darab van
- a szűrést minden szín címkézése előtt megismételjük
- variánsok — `filter(`*`Variáns`*`)` opciók
  - `off`: nincs szűrés
  - `on`: egyszeres szűrés van (alapértelmezés)
  - `repetitive`: mindaddig ismételten szűrünk, amíg az újabb korlátokat eredményez

```
% filter_count_vars(Vars0, Vars, Cnt0, Cnt): Vars0 megszűrve
% Vars-t adja. A megszűrt változók száma Cnt-Cnt0.
filter_count_vars([], [], Cnt, Cnt).
filter_count_vars([V|Vs], Fs, Cnt0, Cnt) :-
        integer(V), !, filter_count_vars(Vs, Fs, Cnt0, Cnt).
filter_count_vars([V|Vs], [V|Fs], Cnt0, Cnt) :-
        (   fd_min(V, Min), Min > 0 -> Cnt1 = Cnt0
        ;   \+ (V = 0) -> V #\= 0, Cnt1 is Cnt0+1
        ;   Cnt1 = Cnt0
        ), filter_count_vars(Vs, Fs, Cnt1, Cnt).
```


Torpedó mintamegoldás — mérési eredmények

Példák

- `fules2a` — Az ismert mezők: 19 átlóbeli mezőről tudjuk, hogy tenger. Egyébként azonos a `fules` feladattal, 6 megoldása van.
- `fules3` — Az ismert mezők: 11 átlóbeli mezőről tudjuk, hogy tenger. 10 megoldása van.

Futási idő (másodperc) — visszalépések száma

Opciók/példa	`fules2a`		`fules3`	
`[]`	7.814	332	50.384	2811
`[-(count_empties)]`	6.766	350	47.696	3248
`[-(count_ones_columns)]`	7.817	334	50.181	2816
`[-(count_ships_occs)]`	15.593	1196	81.247	7543
`[filter([off])]`	14.059	1708	88.031	11236
`[filter([repetitive])]`	8.473	313	54.763	2534
`[relation(clause)]`	8.833	332	55.600	2811
`[diag(ind_arith)]`	7.861	332	50.622	2811
`[diag(reif)]`	8.667	332	56.238	2811
`[label(plain)]`	15.227	1772	90.472	11273
`[label(max_dual)]`	8.315	355	51.739	2688

CLPFD nyomkövetés — háttér

Hanák Dávid, Szeredi Tamás

A SICStus Prolog nyomkövető

A hagyományos nyomkövetési feladatok (pl. spypoint, trace) mellett egyéb, fejlett lehetőségek is elérhetők:

- a program állapota magából a programból is lekérdezhető;
- a töréspontok összetett feltétel-rendszerrel, tág keretek között adhatók meg;
- a töréspontoknál nem kell feltétlenül megállítani a végrehajtást, programrészlet is beszúrható.

A `dispatch_global_fast/4` predikátum

- a beépített globális korlátokat hívja meg;
- utolsó klóza hívja a `dispatch_global/4`-et, paraméterezése azonos vele.
- nem `multifile`-os, ezért lefordítható, tehát gyorsabb;
- a nyomkövetéskor erre a predikátumra teszünk töréspontot.

Az FD halmaz (`fd_set`)

- szigorúan nem összefüggő intervallumok rendezett listája;
- minden intervallumot egy `[From|To]` struktúra ír le;
- pl. `[[1|3], [5|5]] ==> [1,2,3,5]`

Kampók, `portray/1`

Bizonyos események bekövetkezésekor meghívódik egy-egy kampó-predikátum, amely, ha nem hiúsul meg, helyettesíti az alapértelmezett viselkedést, Ilyen pl. a `portray/1`, amely a `print/1,2` használatakor a kiírandó kifejezés minden nem változó részkifejezésére meghívódik.

Alapelvek

Célok

- követhető legyen a véges tartományú korlát (röviden: FD) változók tartományainak szűkülése;
- a programozó értesüljön a korlátok felébredéséről, kilépéséről és hatásairól;
- jól olvasható formában lehessen kiírni FD változókat tartalmazó kifejezéseket.

Fő szolgáltatások

- *Korlátok figyelése*
  - a globális korlátok felébredésekor meghívódik egy predikátum, ez felelős a nyomkövetési információ megjelenítéséért;
  - működik beépített és felhasználói korlátokra is;
  - a meghívott predikátum lehet beépített vagy felhasználó által definiált;
  - az aritmetikai korlátok globálissá fordulnak, így ezek is követhetőek.
- *Kifejezések elnevezése*
  - egyszerű név rendelhető változókhoz;
  - név adható egy tetszőleges kifejezésnek, és ezáltal ennek minden változója is „értelmes” nevet kap;
  - a kifejezés (változó) később a név alapján lekérdezhető;
  - a kiíró eljárások ezekkel a nevekkel hivatkozhatnak a változókra.
- *Tartományok kiírása*
  A nyomkövető eljárásokat biztosít
  - kifejezésekben található FD változók megjelöléséhez;
  - az így módosított kifejezések jól olvasható kiírásához;
  - FD változók listájának *jelmagyarázat-szerű* kiírásához.

Két egyszerű példa

```
| ?- use_module(clpdebug), use_module(library(clpfd)).
| ?- clpdebug_on([legend, log(user)]).
{Advice point for clpfd:dispatch_global_fast/4 added, BID=2}

yes
| ?- clpdebug_name(x, X), X #< 5, X #> 3.
<x>#<5
[max(<x>),exit]
    x = inf..4

<x>#>3
[min(<x>),exit]
    x = {4}

X = 4 ? ;

no
| ?- domain([A,B], 0, 10), sum([A,2,3], #=, B), B #= 6.
scalar_product([1,1,1],[<noname_1>,2,3],#=,<noname_2>)
[max(<noname_1>),min(<noname_2>)]
    noname_1 = 0..5
    noname_2 = 5..10

<noname_2>#=6
[minmax(<noname_2>),exit]
    noname_2 = {6}

scalar_product([1,1,1],[<noname_1>,2,3],#=,6)
[minmax(<noname_1>),exit]
    noname_1 = {1}

A = 1, B = 6 ? ;

no
```


Korlátok figyelése

- a beépített nyomkövető csak az egyszerűbb esetekben kielégítő;
- lehetőség van saját nyomkövető predikátum írására;
- e kettő közül opció segítségével lehet választani.

Predikátumok

- `clpdebug_on`, `clpdebug_on(+Options)`
  Engedélyezi a nyomkövetést alapértelmezett és/vagy megadott beállításokkal.
- `clpdebug_off`
  Kikapcsolja a nyomkövetést.
- `clpdebug:user_info(+Mode, +Constraint, +Port, +Actions)`
  Ez a predikátum hívódik meg, ha nem a beépített nyomkövetőt választjuk. Az innen kiírtak átirányítódnak az opció-listában megadott napló állományba.

Opciók

- `mode(M)` megadja a nyomkövetés módját. `M` lehet
  - `print(legend)`, `print(nolegend)`
    A beépített nyomkövető hívódik meg.
  - egyéb atom
    A `clpdebug:user_info/4` hívódik meg nyomkövetéskor, az első arg. lesz ez az atom.
- `log(L)` a nyomkövetés kimenete L-be irányítódik át. L lehet
  - egy állomány neve
  - `user`: a kimenet a konzolra íródik
  - `none`: a kimenetet eldobjuk

Kifejezések elnevezése

Egy kifejezés elnevezésekor

- a megadott név hozzárendelődik a teljes kifejezéshez;
- a kifejezésben szereplő összes változóhoz egy-egy származtatott név rendelődik – ez a név a megadott névből és a változó szelektorából keletkezik;
- a létrehozott nevek egy globális tárba kerülnek;
- a globális tár mindig egyetlen toplevel híváshoz tartozik (a tár *illékony*).

Predikátumok

- `clpdebug_name(+Name, +Term)`
  A `Term` kifejezéshez a `Name` nevet rendeli az aktuális toplevel hívásban.
- `clpdebug_autoname(+Name, +FileName, +Line, +Selector)`
  Segít, hogy egy változót csak egyszer kelljen elnevezni, ne minden híváskor. Ehhez hivatkozni kell a változó előfordulási helyére, ezért csak állományból konzultált programokhoz használható.
- `clpdebug_noautoname`
  Kitörli az összes automatikus elnevezésre vonatkozó kérést.
- `clptrace`
  - engedélyezi az adott hívásban az *összes* automatikus elnevezés használatát (hatékonysági okokból szükséges);
  - törli a napló állományt.
- `namelist_member(?Name, -Term)`
  - vagy lekérdez egy kifejezést (változót) a globális tárból a neve alapján,
  - vagy felsorolja az összes tárolt név-kifejezés párt.

Tartományok kiírása

A változók tartományának kiírásához és a kifejezésekben található FD változók megjelöléséhez több predikátum adott. Ezeket használja a beépített nyomkövető, de hívhatók kívülről, pl. a `clpdebug:user_info/4`-ből is.

Predikátumok

- `annotate_constraint_vars(+Term0, +Actions, -Term, -Vars)`
  A `Term0` kifejezésben található összes FD változót megjelöli, azaz lecseréli egy `fdvar/3` struktúrára. Ennek tartalma:
  - a változó neve;
  - jelenlegi értékkészlete;
  - értékkészlete az `Actions` akciólista végrehajtása után.

  Az így kapott kifejezés `Term`, a beszúrt `fdvar/3` struktúrák listája `Vars`.
- `print_legend(+Vars)`
  A fent előállított változó listát jelmagyarázatként kiírja:
  - egy sorba egy változó leírása kerül;
  - minden sor elején a változó neve szerepel;
  - a nevet a változó értékkészlete követi.
- `clpdebug:fdvar_portray(+Name, +SetBefore, +SetAfter)`
  Az `annotate_constraint_vars/4` által visszaadott kifejezés kiírásakor alapértelmezésben csak a változók neve íródik ki. Ezzel a kampóval ez a viselkedés felülbírálható.
- `clpdebug:legend_portray(+Name, +SetBefore, +SetAfter)`
  A `print_legend/1` viselkedését felülbíráló kampó, minden változó kiírásakor meghívódik.

Mágikus sorozatok

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

magic(N, L) :-
        length(L, N),
        N1 is N-1,
        domain(L, 0, N1),
        occurrences(L, 0, L),
%       sum(L, #=, N),
%       findall(I, between(0, N1, I), C),
%       scalar_product(C, L, #=, N),
        labeling([ff], L).

occurrences([], _, _).
occurrences([E|Ek], I, List) :-
        exactly(I, List, E), J is I+1,
        occurrences(Ek, J, List).
```

Példa futtatás

```
| ?- magic(4, L).

L = [1,2,1,0] ? ;

L = [2,0,2,0] ? ;

no
| ?- magic(10, L).

L = [6,2,1,0,0,0,1,0,0,0] ? ;

no
```


Példa nyomkövetés

```
| ?- use_module(clpdebug).
{Advice point for user:goal_expansion/3 added, BID=1}

| ?- prolog_flag(source_info, _, on).

| ?- [magic].

| ?- clpdebug_autoname(list, magic, 8, 1).
{Generic advice point added, BID=2}
{Generic advice point, BID=2, disabled (last)}

| ?- clpdebug_on([legend, log('magic.txt')]).
{Advice point for clpfd:dispatch_global_fast/4 added, BID=3}

| ?- clptrace, magic(4, L).
{Generic advice point, BID=2, enabled}
{Generic advice point, BID=2, disabled (last)}

L = [1,2,1,0] ?

yes
```

A `magic.txt` állomány egy részlete

```
exactly(1,[1,2,<list_3>,0],2)
[exit,2=2,<list_3> in_set[[1|1]]]
    list_3 = 1..2 -> {1}

<noname_3>#=1
[max(<noname_3>),exit]
    noname_3 = {1}

exactly(2,[1,2,1,0],1)
[exit,1=1]
```

CHR—Constraint Handling Rules

Fő vonások

- Determinisztikus kifejezés-átíráson alapuló,
- Prolog vagy CLP gazda-megvalósításba beépített
- deklaratív nyelv-kiterjesztés,
- általános, szimbolikus (nem numerikus) **felhasználói** korlátok írására.
- Fő szerző: Thom Früwirth (ECRC, LMU, München)

Példa

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

handler leq.
constraints leq/2.
% X leq Y means variable X is less-or-equal to variable Y

:- op(500, xfx, leq).

reflexivity  @ X leq Y <=> X = Y | true.
antisymmetry @ X leq Y , Y leq X <=> X=Y.
idempotence  @ X leq Y \ X leq Y <=> true.
transitivity @ X leq Y , Y leq Z ==> X leq Z.

| ?- X leq Y, Y leq Z, Z leq X.
% X leq Y, Y leq Z  ----> (transitivity) X leq Z
% X leq Z, Z leq X  <---> (antisymmetry) X = Z
% Z leq Y, Y leq Z  <---> (antisymmetry) Z = Y

Y = X, Z = X ?
```


A CHR szabályok

Szabályfajták

- Egyszerűsítés (Simplification):
  $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j$ `|` $B_1, \ldots, B_k$.
- Propagáció (Propagation):
  $H_1, \ldots, H_i$ `==>` $G_1, \ldots, G_j$ `|` $B_1, \ldots, B_k$.
- Egypagáció (Simpagation):
  $H_1, \ldots, H_l$ `\` H_{l+1}, H_i `==>` $G_1, \ldots, G_j$ `|` $B_1, \ldots, B_k$.

Jelölések

- $H_1, \ldots, H_i$: multi-fej, H_m CHR-korlátok,
- $G_1, \ldots, G_j$: őr (guard), G_m gazda-korlátok
- $B_1, \ldots, B_k$: törzs (body), B_m CHR- vagy gazda-korlátok,
- ahol $i > 0, j \geq 0, k \geq 0, l > 0$.

A szabályok jelentése

- Egyszerűsítés: ha az őr igaz, akkor a (multi-)fej és a törzs ekvivalens.
- Propagáció: ha az őr igaz, akkor a (multi-)fejből kövtkezik a törzs.
- Egypagáció: visszavezethető a fentiekre, mert:
  `Heads1 \ Heads2 <=> Body`
  azonos jelentésű mint
  `Heads1, Heads2 <=> Heads1, Body`,
  csak sokkal hatékonyabb.

A CHR szabályok végrehajtása

Korlátok aktiválása (meghívása vagy fölébresztése)

- Az aktív korláthoz sorra **próbáljuk** az összes szabályt, amelynek fejében előfordul,
- mindegyik fejre **illesztjük** a korlátot (egyirányú egyesítés!)
- többfejű szabályok esetén a korlát-tárban keresünk megfelelő (illeszthető) **partner-**korlátot,
- sikeres illesztés után végrehajtjuk az őr-részt, ha ez is sikeres, a szabály **tüzel**, különben folytatjuk a próbálkozást a következő szabállyal.
- A tüzelés abból áll, hogy kivesszük a tárból a kijelölt korlátokat (egyszerűsítés vagy egypagáció esetén), majd végrehajtjuk a törzset.
- Ha az aktív korlátot nem hagytuk el a tárból, folytatjuk a rá vonatkozó próbálkozást a következő szabállyal.
- Amikor az összes szabályt kipróbáltuk, akkor a korlátot **elaltatjuk**, azaz visszatesszük a tárba (passzív korlátok).

A végrehajtás jellemzői

- A korlátok három állapota: aktív (legfeljebb egy), aktiválható passzív, alvó passzív.
- A korlát akkor válik aktiválhatóvá, amikor egyik változóját **megérintik**, azaz egyesítik egy tőle különböző kifejezéssel.
- Minden alkalommal amikor egy korlát aktívvá válik, az összes rá vonatkozó szabályt végigpróbáljuk.
- A futás akkor fejeződik be, amikor nincs több aktiválható korlát.
- Az őr-részben nem lehet változót érinteni. Az őr-rész két komponense: `Ask & Tell`
  - Ask — változó érintés vagy behelyettesítési hiba meghiúsulást okoz
  - Tell — a felhasználó garantálja, hogy ilyen dolog nem fordul elő

Példa: végeshalmaz-korlátok

```
handler dom_consistency.
constraints dom/2, con/3.
% dom(X,D) var X can take values from D, a ground list
% con(C,X,Y) there is a constraint C between variables X and Y

con(C, X, Y) <=> ground(X), ground(Y) | test(C, X, Y).
con(C, X, Y), dom(X, XD) \ dom(Y, YD) <=>
       reduce(x_y, XD, YD, C, NYD) | new_dom(NYD, Y).
con(C, X, Y), dom(Y, YD) \ dom(X, XD) <=>
       reduce(y_x, YD, XD, C, NXD) | new_dom(NXD, X).

  reduce(CXY, XD, YD, C, NYD):-
       select(GY, YD, NYD1), % try to reduce YD by GD
       (   member(GX, XD), test(CXY, C, GX, GY) -> fail
       ;   reduce(CXY, XD, NYD1, C, NYD) -> true
       ;   NYD = NYD1
       ), !.

    test(x_y, C, GX, GY):- test(C, GX, GY).
    test(y_x, C, GX, GY):- test(C, GY, GX).

  new_dom([], _X) :- !, fail.
  new_dom(DX, X):- DX = [E|RX], dom(X, DX),
         (   RX = [] -> X = E
         ;   true
         ).

% labeling:
constraints labeling/0.

labeling, dom(X, L) #Id <=>
       L = [_,_|_] | member(X, L), dom(X, [X]), labeling
       pragma passive(Id).
```

Az N királynő feladat

```
% an N-queens example:

queens(N, Qs) :-
        length(Qs, N), make_list(1, N, L1_N), domains(Qs, L1_N),
        safe(Qs), labeling.

make_list(I, N, []) :- I > N, !.
make_list(I, N, [I|L]) :-
        I1 is I+1,
        make_list(I1, N, L).

domains([], _).
domains([V|Vs], Dom) :- dom(V, Dom), domains(Vs, Dom).

safe([]).
safe([Q|Qs]) :- no_attack(Qs, Q, 1), safe(Qs).

no_attack([], _, _).
no_attack([X|Xs], Y, I) :-
        con(no_threat(I), X, Y), I1 is I+1,
        no_attack(Xs, Y, I1).

test(no_threat(I), X, Y) :-
        Y =\= X, Y =\= X-I, Y =\= X+I.

| ?- queens(4, Qs).

Qs = [3,1,4,2],                     Qs = [2,4,1,3],
labeling,                           labeling,
dom(2,[2]),                         dom(3,[3]),
dom(4,[4]),                         dom(1,[1]),
dom(1,[1]),                         dom(4,[4]),
dom(3,[3]) ? ;                      dom(2,[2]) ? ;
                                    no
```


Szintaxis

```
Rule           --> [Name @]
                     (Simplification | Propagation | Simpagation)
                        [pragma Pragma].

Simplification --> Heads         <=> [Guard '|'] Body
Propagation    --> Heads         ==> [Guard '|'] Body
Simpagation    --> Heads \ Heads <=> [Guard '|'] Body

Heads          --> Head | Head, Heads
Head           --> Constraint | Constraint # Id
Constraint     --> a callable term declared as constraint
Id             --> a unique variable

Guard          --> Ask | Ask & Tell
Ask            --> Goal
Tell           --> Goal
Goal           --> <<A callable term, including conjunction
                    and disjunction etc.>>

Body           --> Goal

Pragma         --> <<a conjunction of terms usually referring to
                    one or more heads identified via #/2>>
```

Fontosabb pragmák

- `already_in_heads(Id)` — kiküszöböli ugyanazon korlát kivételét és visszarakását
- `passive(Id)` — a hivatkozott fej-korlát csak passzív szerepű lehet.

Példák

Prím-szűrő

```
handler eratosthenes.
constraints primes/1,prime/1.

primes(1) <=> true.
primes(N) <=> N>1 | M is N-1,prime(N),primes(M).

absorb(J) @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

Boole-korlátok — konjunkció

```
handler bool.
constraints and/3, labeling/0.

and(0,X,Y) <=> Y=0.
and(X,0,Y) <=> Y=0.
and(1,X,Y) <=> Y=X.
and(X,1,Y) <=> Y=X.
and(X,Y,1) <=> X=1,Y=1.
and(X,X,Z) <=> X=Z.
and(X,Y,A) \ and(X,Y,B) <=> A=B.
and(X,Y,A) \ and(Y,X,B) <=> A=B.

labeling, and(A,B,C)#Pc <=> label_and(A,B,C), labeling
    pragma passive(Pc).

        label_and(0,X,0).
        label_and(1,X,X).

| ?- and(X, Y, 0), labeling.
  X = 0, labeling ? ;
  X = 1, Y = 0, labeling ? ;
  no
```


Boole-korlátok: számosság

```
constraints card/4.

card(A,B,L):-
        length(L,N), A=<B,0=<B,A=<N, card(A,B,L,N).

triv_sat @ card(A,B,L,N) <=> A=<0,N=<B | true.
pos_sat @ card(N,B,L,N) <=> set_to_ones(L).
neg_sat @ card(A,0,L,N) <=> set_to_zeros(L).
pos_red @ card(A,B,L,N) <=> select(X,L,L1),X==1 |
                A1 is A-1, B1 is B-1, N1 is N-1,
                card(A1,B1,L1,N1).
neg_red @ card(A,B,L,N) <=> select(X,L,L1),X==0 |
                N1 is N-1, card(A,B,L1,N1).
% special cases with two variables
card2nand @ card(0,1,[X,Y],2) <=> and(X,Y,0).
% ...
labeling, card(A,B,L,N)#Pc <=>
  label_card(A,B,L,N), labeling
    pragma passive(Pc).

label_card(A,B,[],0):- A=<0,0=<B.
label_card(A,B,[0|L],N):- N1 is N-1, card(A,B,L,N1).
label_card(A,B,[1|L],N):-
    A1 is A-1, B1 is B-1, N1 is N-1, card(A1,B1,L,N1).

| ?- card(2,3,L), labeling.

L = [1,1], labeling ? ;

L = [0,1,1] ? ; L = [1,0,1] ? ; L = [1,1,_A] ? ;
L = [0,0,1,1] ? ; L = [0,1,0,1] ? ; L = [0,1,1,_A] ? ;
% ...
```

A Mercury nagyhatékonyságú LP megvalósítás

Célok

- Nagybani programozás támogatása
- Produktivitás, megbízhatóság, hatékonyság növelése

Eszközök, elvek

- Teljesen deklaratív programozás
- Funkcionális elemek integrálása
- Hagyományos (Prolog) szintaxis megőrzése
- Típus, mód és determinizmus információk használata
- Szeparált fordítás támogatása
- Prologénál erősebb modul-rendszer
- Sztenderd könyvtár

Elérhetőség

- Fejlesztő (nyelv+implementáció): University of Melbourne
- `http://www.cs.mu.oz.au/mercury/`
- GPL

Mercury példaprogram

File-név illesztés

- A feladat: operációs rendszerek file-név-illesztéséhez hasonló funkció megvalósítása.

Adott minta és karaktersorozat illesztésekor

- A `?` egy tetszőleges karakterrel illeszthető.
- A `*` egy tetszőleges (esetleg üres) karakter-sorozattal illeszthető.
- A `\`*c* karakter-pár a *c* karakterrel illeszthető, ha egy minta `\`-re végződik, az illesztés meghiúsul.
- Bármely más karakter csak önmagával illeszthető.

A Mercury program hívási formája:

`match Pattern1 Name Pattern2`

Itt a `Pattern1` és `Pattern2` mintákban a `*` és `?` azonos elrendezésben kell előfordulniuk.

A program funkciója

- a `Pattern1` mintára (az összes lehetséges módon) illeszti a `Name` nevet,
- a `*` és `?` karakterek helyébe kerülő szövegeket a `Pattern2` mintába behelyettesíti,
- és az így kapott neveket kiírja.

A file-név-illesztő Mercury program listája

```
:- module match.
/*-----------------------------------------------------*/
:- interface.

:- import_module io.
:- pred main(io__state::di, io__state::uo) is det. % kötelező

/*-----------------------------------------------------*/
:- implementation.
:- import_module list, std_util, string, char.

main -->
    command_line_arguments(Args),
    (   {Args = [P1,N1,P2]} ->
         {solutions(match(P1, N1, P2), Sols)},
         format("Pattern `%s' matches `%s' as `%s'\
 matches the following:\n\n",
                        [s(P1),       s(N1),  s(P2)]),
         write_list(Sols, "\n", write_string),
         write_string("\n*** No (more) solutions\n")
    ;   write_string("Usage: match <p1> <n1> <p2>\n")
    ).
```


Példaprogram, folytatás

```
:- pred match(string::in, string::in, string::in,
              string::out) is nondet. % szükséges
match(Pattern1, Name1, Pattern2, Name2) :-
    to_char_list(Pattern1, Ps1),
    to_char_list(Name1, Cs1),
    to_char_list(Pattern2, Ps2),
    match_list(Ps1, Cs1, L),
    match_list(Ps2, Cs2, L),
    from_char_list(Cs2, Name2).

:- type subst ---> any(list(char)) ; one(char).

:- pred match_list(list(char), list(char), list(subst)).
:- mode match_list(in, in, out) is nondet. % mindkettő,
:- mode match_list(in, out, in) is nondet. % vagy egyik se
match_list([], [], []).
match_list([?|Ps], [X|Cs], [one(X)|L]) :-
    match_list(Ps, Cs, L).
match_list([*|Ps], Cs, [any(X)|L]) :-
    append(X, Cs1, Cs),
    match_list(Ps, Cs1, L).
match_list([\, C|Ps],  [C|Cs], L) :-
    match_list(Ps, Cs, L).
match_list([C|Ps], [C|Cs], L) :-
    C \= (*), C \= ?, C \= (\),
    match_list(Ps, Cs, L).
```

Típusok

A típusok fajtái

- primitív: `char`, `int`, `float`, `string`
- predikátum: `pred`, `pred(T)`, `pred(T1, T2)`, ...
- függvény: `(func) = T`, `func(T1) = T`, ...
- univerzális: `univ`
- „a világ állapota”: `io__state`
- felhasználó által bevezetett

Felhasználói típusok

- megkülönböztetett unió (SML: `datatype`)
- ekvivalencia (típusátnevezés) (SML: `type`)
- absztrakt

Megkülönböztetett unió

- Enumerációs és rekord típus
- lehet monomorf vagy polimorf

Enumeráció típus

```
:- type fruit ---> apple ; orange ; banana ; pear.
```

Rekord típus

```
:- type itree ---> empty ; leaf(int) ; branch(itree, itree).
```

Polimorfikus típus

```
:- type pair(T1, T2) ---> T1 - T2.
```

Trükkös

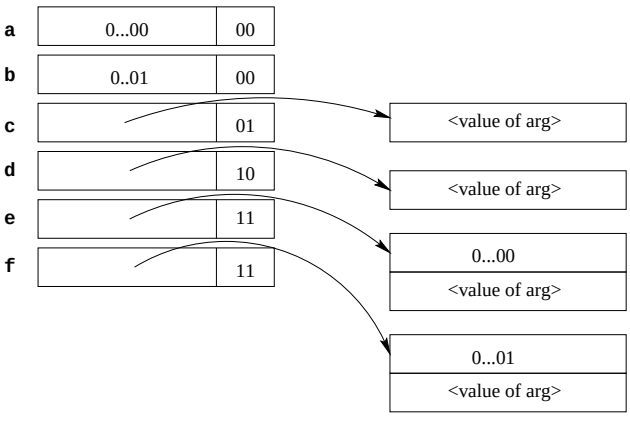
```
:- type tricky ---> { int ; int } ; { { int } }.
```

A játékszabályok

- `:- type` ⟨típus⟩ `--->` ⟨törzs⟩`.`
- a ⟨törzs⟩ minden konstruktorában az argumentumok típusok vagy változók
- a ⟨törzs⟩ minden változójának szerepelnie kell ⟨típus⟩-ban
- ⟨típus⟩ változói különbözők
- a típusok között névekvivalencia van
- egy típusban nem fordulhat elő egynél többször azonos nevű és argumentumszámú konstruktor

Megkülönböztetett unió implementációja

```
:- type t ---> a ; b ; c(t1) ; d(t2) ; e(t3) ; f(t4).
```



Következmények

- egyszerű típusok általában „dobozolatlanul” implementálhatók
- „heterogén” kolleckció esetében explicit csomagolásra van szükség
- kevés (4) típus esetén nem drágább a csomagolás

Más típusú típusmegadások

Ekvivalencia típus

- `:- type` ⟨típus⟩ `==` ⟨típus⟩`.`
- `:- type assoc_list(K, V) == list(pair(K, V)).`
- nem lehet ciklikus
- a jobb és a bal oldal ekvivalens

Absztrakt típus

- `:- type` ⟨típus⟩`.`
- `:- type t2(T1, T2).`
- a definíció el van rejtve az implementációs részben

A típusok használata

- A predikátumok és függvények argumentumainak meg kell mondani a típusát.
- `:- pred is_all_uppercase(string).`
- `:- func length(list(T)) = int.`

A típushozzárendelés

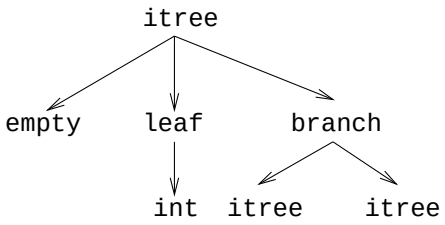
- típus rendelése változókhoz, konstruktorokhoz, függvényekhez, eljárásokhoz
- a fordító automatikusan végzi a megadott típusinfo alapján
- akkor sikerül, ha létezik egyértelmű legáltalánosabb hozzárendelés

Módok, behelyettesítettség

Mód

- két behelyettesítettségi állapotból álló pár
- az első állapot arról szól, ahogy a paraméter bemegy, a második arról, ahogy kijön egy adott függvényből
- pl.: `out`: (szabad) változó megy be, tömör kifejezés jön ki

A behelyettesítettségi fa



- Az állapot leírásakor a típust tartalmazó („vagy”) csúcsokhoz rendelünk behelyettesítettségi állapotot.
- Deklarációban a `bound/1`, a `free/0` és a `ground/0` funktorokat használhatjuk.
- `:- inst bs = bound(empty; leaf(free); branch(bs,bs)).`
- Parametrizált `inst`-eket is csinálhatunk:
  `:- inst bs(Inst) = bound(empty; leaf(Inst); branch(bs(Inst),bs(Inst))).`

Módok használata

Mód-deklaráció

- Módok definiálása:
  `:- mode` ⟨m⟩ `::` ⟨inst1⟩ `->` ⟨inst2⟩`.`

  ```
  :- mode in :: ground -> ground.
  :- mode out :: free -> ground.
  ```

- Módok átnevezése:
  `:- mode` ⟨m1⟩ `::` ⟨m2⟩`.`

  ```
  :- mode (+) :: in.
  :- mode (-) :: out.
  ```

- Parametrizált módok:
  ```
  :- mode in(Inst) ::  Inst -> Inst.
  :- mode out(Inst) ::  free -> Inst.
  ```

Predikátum-mód deklaráció

- Egy eljárás minden paraméteréről megmondjuk milyen módú.

  ```
  :- pred append(list(T), list(T), list(T)).
  :- mode append(in, in, out).
  :- mode append(out, out, in).
  ```

- Egyetlen mód esetén összevonható a `pred` deklarációval.
- Függvényeknek is lehet több módja.
- Mercuryban egy adott predikátum egy adott módját nevezzük eljárásnak.

Módok: mire kell figyelni?

- `free` változókat még egymással sem lehet összekapcsolni,

  ```
  :- mode append(in(listskel(free)),
       in(listskel(free)),
       out(listskel(free))).
  ```

  hibás!
- Ha egy predikátumnak nincs predikátum-mód deklarációja, akkor a fordító kitalálja az összes szükségeset (`--infer-all` kapcsoló),
- de függvényeknél ilyenkor felteszi, hogy minden argumentuma `in` és az eredménye `out`.
- A fordító átrendezi a hívásokat, hogy a mód korlátokat kielégítse: ha ez nem megy, hibát jelez. (Jobbrekurzió!)
- A megadottnál „jobban” behelyettesített argumentumokat egyesítésekkel kiküszöböli a fordító. Ezeket a módokat le se kell írni (de érdemes lehet.) Példa: `:- mode append(in, out, in).` a szétszedő `append`-et fogja használni.
- A jelenlegi implementáció nem kezeli a részlegesen behelyettesített adatokat.

Determinizmus

Determinizmus kategóriák

Minden predikátum minden módjára (azaz minden eljárásra) megadjuk, hogy hányféleképpen sikerülhet, és hogy meghiúsulhat-e.

A kategóriák nevei

meghiúsulás\megoldások	0	1	> 1
nem	`erroneous`	`det`	`multi`
igen	`failure`	`semidet`	`nondet`

A determinizmus-deklaráció

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
:- mode append(in, in, in) is semidet. % nem szükséges
```

Összevont deklaráció

```
:- pred p(int::in) is det.
p(_).
```

„Egzotikus” determinizmusok

- `failure` determinizmusú a `fail/0`
- `erroneous` determinizmusú a `require__error/1`

Függvények determinizmusa

- Ha minden argumentuma bemenő, akkor a determinizmusa csak `det`, `semidet`, `erroneous` vagy `failure` lehet.
- Ha nem így lenne, akkor az matematikai értelemben nem lenne függvény.
- Pl. `between(in, in, out)` nem írható függvényalakban.

Példák

Helyesek-e?

```
:- type fruit ---> banana ; orange ; lemon ; grape.
:- type ice_cream ---> lemon ; banana ; orange.
:- type unsi ---> n ; s(unsi).
```

Milyen módjai vannak és milyen a determinizmusa?

```
:- pred make_ice_cream(fruit, ice_cream).
make_ice_cream(lemon,  lemon).
make_ice_cream(orange, lemon).
make_ice_cream(banana, banana).

:- func factorial(int) = int.
factorial(N) = F :-
    (   N = 0 -> F = 1
    ;   N > 0 -> F = factorial(N-1)*N
    ;   error("out of domain")
    ).

:- pred even(num).
even(z).
even(s(N)) :-
    odd(N).

:- pred odd(num).
odd(s(N)) :-
    even(N).
```


Problémák a determinizmussal

- `det` és `semidet` módú eljárásokból nem hívható `nondet` vagy `multi` eljárás
- például a `main/2` eljárás `det` módú

Megoldások

- az összes megoldást megkeressük: `std_util__solutions/2`
- csak egy megoldást akarunk (és nem érdekes melyik)
  - ha az eljárás kimenő változóit nem használjuk fel, akkor az első utáni megoldásokat levágja a rendszer: `member(1, [1,1])`
  - kihasználjuk, hogy sosem fogunk egynél több megoldást keresni (committed choice nondeterminism): `cc_nondet`, `cc_multi` determinizmus
- (néhány megoldást keresünk meg: `std_util__do_while/4`)

Amire még nincs igazi megoldás

- ha `nondet` eljárásban akarunk meghívni egy `nondet` eljárást, amelynek minden megoldása ekvivalens
- tervezett megoldás: `unique [X] goal(X)`
- addig a C interfésszel kell trükközni

Problémák a determinizmussal, példa

Feladat

1. Soroljuk fel egy halmaz összes részhalmazát!
2. Minden megoldást pontosan egyszer adjunk ki!

```
:- module resze.

:- interface.
:- import_module io.

:- pred main(io__state::di, io__state::uo) is cc_multi.

:- implementation.
:- import_module int, set, list, std_util.

main -->
    read_int_listset(L, S),
    io__write_string("The set version:\n"),
    {std_util__unsorted_solutions(resze(S), P)},
    io__write_list(P, " ", io__write),
    io__write_string("\n\nThe list version:\n"),
    {std_util__unsorted_solutions(lresze(L), PL)},
    io__write_list(PL, " ", io__write), io__nl.

:- pred read_int_listset(list(int)::out, set(int)::out,
          io__state::di, io__state::uo) is det.
read_int_listset(L, S) -->
    io__read(R),
    {   R = ok(LO) -> L = LO, set__list_to_set(L, S)
    ;   set__init(S), L = []
    }.
```

Problémák a determinizmussal, folytatás

1. megoldás: `set` absztrakt adattipussal
A `set__member/2` felsoroló jellege miatt nem teljesíti a 2. feltételt.

```
:- pred resze(set(T)::in, set(T)::out) is multi.
resze(A, B) :-
   set__init(Fix),
   resze(A, B, Fix).

:- pred resze(set(T)::in, set(T)::out, set(T)::in) is multi.
resze(A, B, Fix) :-
   (   set__member(X, A)
   ->  set__delete(A, X, A1),
       (   resze(A1, B, Fix)
       ;   resze(A1, B, set__insert(Fix, X))
       )
   ;   B = Fix
   ).
```

2. megoldás: `list` adattipussal
A lista fejének levágása (szemi)determinisztikus, így teljesül a 2. feltétel.

```
:- pred lresze(list(T)::in, list(T)::out) is multi.
lresze(A, B) :-
   lresze(A, B, []).

:- pred lresze(list(T)::in, list(T)::out, list(T)::in) is multi.
lresze(A, B, Fix) :-
   (   A = [X|A1],
       (   lresze(A1, B, Fix)
       ;   lresze(A1, B, [X|Fix])
       )
   ;   A = [], B = Fix
   ).
```


Problémák a determinizmussal, folytatás

Egy példa-fordítás és futtatás

```
benko:~/mercury$ ls resze*
resze.m
benko:~/mercury$ mmake resze.dep
mmc --generate-dependencies     resze
benko:~/mercury$ mmake resze
rm -f resze.c
mmc --compile-to-c --grade asm_fast.gc resze.m > resze.err 2>&1
mgnuc --grade asm_fast.gc -c resze.c -o resze.o
c2init --grade asm_fast.gc resze.c > resze_init.c
mgnuc --grade asm_fast.gc -c resze_init.c -o resze_init.o
ml --grade asm_fast.gc -o resze resze_init.o resze.o
benko:~/mercury$ ls resze*
resze    resze.d    resze.dv   resze.m  resze_init.c
resze.c  resze.dep  resze.err  resze.o  resze_init.o
benko:~/mercury$ ./resze
[1, 2].
The set version:
[1, 2] [2] [1] [] [1, 2] [1] [2] []

The list version:
[2, 1] [1] [2] []
benko:~/mercury$
```


Committed choice nondeterminism

Használat

- `cc_multi` a `multi` helyett
- `cc_nondet` a `nondet` helyett
- olyan helyeken használhatjuk, ahol biztosan nem lesz szükségünk több megoldásra
- két predikátummód-deklaráció különbözhet csak a `cc`-s mivoltukban

```
:- mode append(out, out, in) is multi.
:- mode append(out, out, in) is cc_multi.
```

Haszna

- I/O műveletek csak `det` és `cc_multi` eljárásokban lehetségesek
- hatékonyság
- nemkanonikus ábrázolású adattípusok egyenlőségének eldöntése (többet később)

Egy `cc_multi`-s példa

```
:- module queens.

:- interface.
:- import_module list, int, io.

:- pred main(state::di, io__state::uo) is cc_multi.

:- implementation.

main -->
   (   {queen([1,2,3,4,5,6,7,8], Out)} -> write(Out)
   ;   write_string("No solution")
   ), nl.

:- pred queen(list(int)::in, list(int)::out) is nondet.
queen(Data, Out) :-
   perm(Data, Out),
   safe(Out).

:- pred safe(list(int)::in) is semidet.
safe([]).
safe([N|L]) :-
   nodiag(N, 1, L),
   safe(L).

:- pred nodiag(int::in, int::in, list(int)::in) is semidet.
nodiag(_, _, []).
nodiag(B, D, [N|L]) :-
   D \= N-B, D \= B-N,
   nodiag(B, D+1, L).
```

Egyszeres hivatkozású (unique) módok

Jellemzők

- Az adott paraméterre csak egy referencia lehet.
- A referencia megszűntével a memória felszabadítható vagy újrahasznosítható.
- Segítségével destruktív frissítést valósítható meg.
- Ezt használja pl. az `io` könyvtár is.

Új behelyettesítettségek

- `unique`: olyan, mint `ground`, de csak egyszeres hivatkozás lehet
- `unique(...)`: olyan, mint `bound(...)`, de csak egyszeres hivatkozás lehet
- `dead`: nincs rá több hivatkozás

Sztenderd módok

- `:- mode uo :: free -> unique.`
- `:- mode ui :: unique -> unique.`
- `:- mode di :: unique -> dead.`

A jelenlegi implementáció korlátai

- csak a legfelső szinten megengedett a `unique` behelyettesítettség
- a memória újrahasznosítása csak az `io` és az `array` könyvtárakban működik

Egy gráf összefüggőségének tesztelése

```
:- type graph(T) == map(T, list(T)).
:- type igraph == graph(int).
:- type ba == array(bool).

:- pred connected(igraph::in, int::in) is semidet.
connected(G, N) :-
    (   N = 0 -> true
    ;   init(N, no, V),
        traverse(G, 0, 0, N, V, _)
    ).

:- pred traverse(igraph::in, int::in, int::in, int::out,
      ba::array_di, ba::array_uo) is det.
traverse(G, N, C0, C, V0, V) :-
    (   lookup(V0, N, yes)
    ->  C = C0, V = V0       % already visited
    ;   search(G, N, Ns)
    ->  set(V0, N, yes, V1), % not visited yet, has children
        traverse_list(Ns, G, C0+1, C, V1, V)
    ;   C = C0+1,
        set(V0, N, yes, V)   % not visited, no children
    ).

:- pred traverse_list(list(int)::in, igraph::in,
           int::in, int::out,
           ba::array_di, ba::array_uo) is det.
traverse_list([], _, C, C) --> [].
traverse_list([N|Ns], G, C0, C) -->
    traverse(G, N, C0, C1),
    traverse_list(Ns, G, C1, C).
```


Visszaléptethető destruktív frissítés

Jobbára egyszeres hivatkozású módok

- egyszeres hivatkozású érték, de visszalépéskor vissza tud állni
- implementáció pl. „trail" segítségével
- új módok

```
:- mode muo :: free -> mostly_unique.
:- mode mui :: mostly_unique -> mostly_unique.
:- mode mdi :: mostly_unique -> mostly_dead.
```


Magasabb rendű eljárások

Részlegesen paraméterezett eljárások

- segédeszközök: `call/2`, `call/3`, ... eljárások
- a `call/<I>` eljárások Mercuryban beépítettek

A `call/4` eljárás Prolog definíciója

```
% Pred az A, B és C utolsó argumentumokkal
% meghívva igaz.
call(Pred, A, B, C) :-
    Pred =.. FArgs,
    append(FArgs, [A,B,C], FArgs3),
    Pred3 =.. FArgs3, call(Pred3).
```

Példa: a map eljárás definíciója

```
% map(Pred, Xs, Ys): Az Xs lista elemeire
% a Pred transzformációt alkalmazva kapjuk az Ys listát.
:- pred map(pred(X, Y), list(X), list(Y)).
:- mode map(pred(in, out) is det, in, out) is det.
:- mode map(pred(in, out) is semidet, in, out) is semidet.
:- mode map(pred(in, out) is multi, in, out) is multi.
:- mode map(pred(in, out) is nondet, in, out) is nondet.
:- mode map(pred(in, in) is semidet, in, in) is semidet.
map(P, [H|T], [X|L]) :-
    call(P, H, X),
    map(P, T, L).
map(_, [], []).

:- import_module int.

:- pred negyzet(int::in, int::out) is det.
negyzet(X, X*X).

:- pred p(list(int)::out) is det.
p(L) :-
    map(negyzet, [1,2,3,4], L).

:- pred p1(list(int)::out) is det.
p1(L) :-
    map((pred(X::in, Y::out) is det :- Y = X*X), [1,2,3,4], L).
```


Magasabb rendű kifejezések létrehozása

Magasabbrendű eljárások

- tegyük fel, hogy létezik

  ```
  :- pred sum(list(int)::in, int::out) is det.
  ```

- λ-kifejezéssel:

  ```
  X = (pred(Lst::in, Len::out) is det :- sum(Lst, Len))
  ```

- az eljárás nevét használva (a nevezett dolognak csak egyféle módja lehet és nem lehet 0 aritású függvény):

  ```
  Y = sum
  ```

- `X` és `Y` típusa: `pred(list(int), int)`

Magasabbrendű függvények

- ha adott

  ```
  :- func mult_vec(int, list(int)) = list(int).
  ```

- λ-kifejezéssel:

  ```
  X = (func(N, Lst) = NLst :- NLst = mult_vec(N, Lst))
  Y = (func(N::in, Lst::in) = (NLst::out) is det
     :- NLst = mult_vec(N, Lst))
  ```

- a függvény nevét használva:

  ```
  Z = mult_vec
  ```


Többargumentumú megasabbrendű kifejezések (currying)

Eljárások és függvények

- `Sum123 = sum([1,2,3])`: `Sum123` típusa `pred(int)`
- `Double = mult_vec(2)`: `Double` típusa `func(list(int)) = list(int)`

DCG

- Külön szintaxis az olyan eljárásokra, amelyek egy akkumulátor párt használnak
- Példa (típusa `pred(list(string), int, io__state, io__state)`):

  ```
  Pred = (pred(Strings::in, Num::out, di, uo) is det -->
      io__write_string("The strings are: "),
      { list__length(Strings, Num) },
      io__write_strings(Strings),
      io__nl
  )
  ```

Amire figyelni kell

- beépített nyelvi konstrukciókat nem lehet „curryzni”
- ilyenek pl.: `=`, `\=`, `call`, `apply`
- `list__filter([1,2,3], \=(2), List)` helyett:
  `list__filter([1,2,3], (pred(X::in) is semidet :- X \= 2), List)`

Magasabbrendű kifejezések meghívása

Eljárások meghívása

- `call(Closure, Arg`$_1$`, ..., Arg`$_n$`)`, $n \geq 0$
- `solutions(match(P1, N1, P2), Sols)`

Függvények meghívása

- `apply(Closure2, Arg`$_1$`, ..., Arg`$_n$`)`, $n \geq 0$
- `List = apply(Double, [1,2,3])`

Magasabbrendű módok

Mód és determinizmus

- A magasabbrendű kifejezések determinizmusa a módjuk része (és nem a típusuké).
- Például:

```
:- pred map(pred(X, Y), list(X), list(Y)).
:- mode map(pred(in, out) is det, in, out) is det.
```

Beépített behelyettesítettségek

- Eljárások:
  `pred(`⟨mode$_1$⟩`, ..., `⟨mode$_n$⟩`) is` ⟨determinism⟩, ahol $n \geq 0$
- Függvények:
  `(func) =` ⟨mode⟩ `is` ⟨determinism⟩
  `func(`⟨mode$_1$⟩`, ..., `⟨mode$_n$⟩`) =` ⟨mode⟩ `is` ⟨determinism⟩, ahol $n > 0$

Beépített módok

- A nevük megegyezik a behelyettesítettségek nevével, és a pár mindkét tagja ugyanolyan, a névnek megfelelő behelyettesítettségű.
- Egy lehetséges definíció lenne:

```
:- mode (pred(Inst) is Det) :: in(pred(Inst) is Det).
```

Amire figyelni kell

- Magasabbrendű kimenő paraméter:

```
:- pred foo(pred(int)).
:- mode foo(free -> pred(out) is det) is det.
foo(sum([1,2,3])).
```

- Magasabbrendű kifejezések nem egyesíthetők:
  `foo((pred(X::out) is det :- X = 6))` hibás.

Modul-rendszer

Támogatott tulajdonságok

- szeparált fordítás
- modulok egymásbaágyazása
- absztrakt típusok használata

Deklarációk

- modul kezdés: `:- module` ⟨modulename⟩.
- interfész: `:- interface.`
- megvalósítás: `:- implementation.`
- lezárás (opcionális): `:- end_module` ⟨modulename⟩.

Az interfész rész

- Minden szerepelhet, kivéve függvények, predikátumok és almodulok definíciója.
- Az itt szereplő dolgok fognak kilátszani a modulból.

Az implementációs rész

- Szerepelnie kell a függvények, predikátumok, absztrakt típusok és almodulok definíciójának.
- Az itt deklarált dolgok lokálisak a modulra.

Modul-rendszer, folytatás

Más modulok felhasználása

- `:- import_module` ⟨modules⟩.
  Ezután nem szükséges modulkvalifikáció.
- `:- use_module` ⟨modules⟩.
  Csak explicit modulkvalifikációval használhatjuk fel a benne levő dolgokat.

Modulkvalifikáció

- ⟨mod⟩`:`⟨submodule⟩`:...:`⟨submodule⟩`:`⟨name⟩
- Egyelőre a `:` helyett a `__` javasolt, mert lehet, hogy később a `.` lesz a modulkvalifikátor és a `:` típuskvalifikátor.

Almodulok

- beágyazott almodulok: a főmodul fájljában definiált
- szeparált almodulok: külön fájlban definiált
- a jelenlegi implementációnál az előbbi használata problémás