Nagyhatékonyságú logikai programozás

Jegyzetek a BME informatikus hallgatói számára

Kézirat

Szeredi Péter, Benkő Tamás

Számítástudományi
és Információelméleti Tanszék

IQSOFT Rt.

{szeredi,benko}@iqsoft.hu

A jegyzetet az előadásvázlatok alapján készítette:

Nepusz Tamás

tamas@eet.bme.hu

Javítások:

Szárnyas Gábor

szarnyas@mit.bme.hu

Budapest, 2002–2017

Tartalomjegyzék

1. fejezet

Constraint Logic Programming (CLP)

A *CLP* (*Constraint Logic Programming*) a logikai programozás egy új irányzata. Alapvető tulajdonsága, hogy a program tartalmazhat változók értékeire való megszorításokat, *korlát*okat (*constraint*). Az alábbi táblázatban összefoglaljuk a legelterjedtebb logikai programozási nyelv, a Prolog alapelemeit és ezek CLP megfelelőit.

Prolog	CLP
hívás	hívás vagy constraint
egyesítés	korlátmegoldás (constraint solving)
válasz-behelyettesítés	válasz-korlát

A továbbiakban a CLP séma és a Prolog nyelv ötvözéséből keletkezett programozási eszközökkel fogunk foglalkozni.

1.1. A CLP nyelv elemei

Minden CLP megvalósítás egy $\mathcal{X}$ adattartományon és az ezen értelmezett korlátokra (relációkra) vonatkozó „erős" következtetési mechanizmus. Az $\mathcal{X}$ adattartomány különféle megválasztásaiból más-más CLP sémák adódnak. Néhány példa:

- $\mathcal{X} = \mathbb{R}$ vagy $\mathbb{Q}$ (racionális vagy valós számok).
  Itt korlátoknak tekinthetjük a racionális vagy valós számok közt fennálló lineáris egyenlőségeket egyenlőtlenségeket, következtetési mechanizmusnak pedig a Gauss-eliminációt és a szimplex módszert.
- $\mathcal{X} = \mathrm{FD}$ (egész számok véges tartománya, angolul FD = Finite Domain).
  Korlátoknak vehetjük a különféle aritmetikai és kombinatorikai relációkat, következtetési mechanizmusnak pedig a mesterséges intelligencia kutatásokból ismert CSP (Constraint Solving Problem, korlátkielégítési probléma) módszereket.
- $\mathcal{X} = \mathbb{B}$ (logikai igaz és hamis értékek).
  Itt a korlátok a predikátum kalkulusban fennálló relációk, a következtetési mechanizmus pedig szintén a mesterséges intelligencia területéhez tartozó SAT (satisfiability, Boole-kielégíthetőség) módszer.

A függvényeket és a relációkat CLP-ben – a Prolog-tól eltérően – nem szintaktikusan, hanem szemantikusan kezeljük, a Prologban a program dolga, hogy jelentést tulajdonítson egy függvény (struktúra) - kifejezésnek (pl. az `is/2` predikátum). A Prolog csak a szintaktikus egyesítést (a `=/2` műveletet) ismeri, ami például az `X+2=Y+3` kifejezést sikertelenül próbálja feldolgozni, így a hívás meghiúsul. Ha ezt a kifejezést egy alaphalmaz, domain felett nézzük, pl. a valós számok fölött (CLP(R)), akkor azt a matematikailag helyes megoldást kapjuk, hogy `X=Y+1`. Ahhoz, hogy ezt az eredményt kaphassuk, szükség van a *korlátmegoldó*ra (*constraint solver*), ami a változókból, értékekből, függvényekből és a relációkból álló korlátokat ki tudja értékelni, és ellenőrizni tudja a konzisztenciájukat. Ha egy új korlát hozzávételével a tár inkonzisztenssé válna, azt a korlátmegoldó észreveszi, és a programnak az aktuális végrehajtási ága meghiúsul. Ilyenkor a Prolog végrehajtás szabályai szerint visszalépés következik be.

A fent elmondottak formalizálva a következőképpen néznek ki:

Egy CLP rendszer egy $\langle\mathcal{D},\mathcal{F},\mathcal{R},\mathcal{S}\rangle$ struktúrával írható le, ahol az egyes elemek jelentése:

- $\mathcal{D}$ – egy tartomány, pl. a valós számok ($\mathbb{R}$), a racionális számok ($\mathbb{Q}$), az egész számok ($\mathbb{N}$), a Boole-értékek ($\mathbb{B}$), karakterfüzérek, listák, Prolog végrehajtási fák (Herbrand-fák, $\mathbb{H}$) tartománya
- $\mathcal{F}$ – a fenti tartományon értelmezett függvények halmaza, pl. $+,-,*,\wedge,\vee$
- $\mathcal{R}$ – a fenti tartományon értelmezett relációk halmaza, pl. $=,\neq,<,>,\in$
- $\mathcal{S}$ – egy korlátmegoldó algoritmus $\langle\mathcal{D},\mathcal{F},\mathcal{R}\rangle$-re, azaz a $\mathcal{D}$ tartományon értelmezett $\mathcal{F}\cup\mathcal{R}$ halmazbeli jelekből felépített korlátokra

1.2. CLP szintaxis és deklaratív szemantika

Szintaxis:

program:	klózok halmaza
klóz:	`P:-` G_1, …, G_n, ahol G_i vagy cél vagy constraint
deklaratív olvasata:	`P` igaz, ha G_1, …, G_n mind igaz
kérdés:	`?-` G_1 , …, G_n
válasz egy kérdésre:	korlátoknak egy olyan konjunkciója, amelyből a kérdés következik

1.3. CLP procedurális szemantika

A CLP séma szemantikája leírható, mint egy kezdeti célból történő levezetés, amely felhasználja a program klózait. A levezetés egy állapota egy $\langle G,s\rangle$ párral jellemezhető, ahol:

- G a megoldandó célok és korlátok konjunkciója
- s egy *korlát-tár*, amely az eddig felhalmozott korlátokat tartalmazza. A korlát-tár tartalma mindig *kielégíthető*, kezdetben pedig üres.

Sok CLP megvalósításban a korlát-tár csak a korlátok egy bizonyos osztályát tárolhatja, ezeket a korlátokat *egyszerű korlát*oknak nevezzük, a korlát-tárba be nem tehető korlátokat pedig *összetett korlát*oknak. Például a SICStus `clpfd` könyvtárának használatakor az egyszerű korlátok csak az $\in$ relációt tartalmazhatják. Az összetett korlátok felfüggesztve, *démon*ként várnak arra, hogy a korlátmegoldónak segíthessenek. A továbbiakban az egyszerű korlátokat kisbetűkkel (pl. c), az általános jellegű korlátokat pedig nagybetűkkel (pl. C) jelöljük.

Procedurális értelmezésben egy `P :- G`$_1$`, G`$_2$`, …, G`$_n$ klóz jelentése a következő: `P` elvégzéséhez el kell végezni `G`$_1$-et, `G`$_2$-t, …`G`$_n$-et.

A végrehajtás alaplépése:

- A végrehajtandó cél egy részcéljának, `P`-nek determinisztikus kiválasztása
- Egy `P`-re illeszkedő klóz, `P'` nemdeterminisztikus kiválasztása
- `P'` végrehajtása és a korlát-tár konzisztenciájának ellenőrzése
- Ha a korlát-tár konzisztens, akkor az eljárás folytatható a következő részcéllal, ha viszont nem konzisztens, akkor a végrehajtás ezen ága meghiúsul, visszalépés következik be, melynek során a hagyományos visszalépési mechanizmus mellett a korlát-tár tartalmát is vissza kell fejteni a legutóbbi választási pontig.

A végrehajtás minden $\langle G, s\rangle$ állapotában teljesül, hogy s konzisztens, és $G \wedge s \Rightarrow Q$, ahol Q a kezdő kérdés. A végrehajtás akkor áll meg, ha egy olyan $\langle G_c, s_c\rangle$ állapotba kerültünk, ahol G_c-re már egyetlen következtetési lépés sem végezhető el. Ekkor a végrehajtás eredménye az s_c korlát-tár (vagy annak egy adott változóhalmazra való vetítése a többi változó egzisztenciális kvantifikálásával), valamint az esetlegesen fennmaradó G_c korlátok.

Általános következtetési lépések

- *rezolúció*:
  $\langle \mathtt{P}\ \&\ \mathtt{G},\ s\rangle \Rightarrow \langle \mathtt{G}_1\ \&\ \dots\ \&\ \mathtt{G}_n\ \&\ \mathtt{G},\ \mathtt{P} = \mathtt{P}' \wedge s\rangle$, feltéve, hogy a programban van egy $\mathtt{P}'$ `:-` $\mathtt{G}_1, \dots,$ $\mathtt{G}_n$ klóz.
- *korlát-megoldás*:

  $\langle \mathtt{c}\ \&\ \mathtt{G},\ s\rangle \Rightarrow \langle \mathtt{G},\ s \wedge \mathtt{c}\rangle$, tehát egy egyszerű korlát bekerülhet a korlát-tárba, feltéve, ha a korlát-tár továbbra is konzisztens marad
- *korlát-erősítés*:

  $\langle \mathtt{C}\ \&\ \mathtt{G},\ s\rangle \Rightarrow \langle \mathtt{C}'\ \&\ \mathtt{G},\ s \wedge\ \mathtt{c}\rangle$ha s-ből következik, hogy `C` ekvivalens ($\mathtt{C}' \wedge \mathtt{c}$)-vel ($\mathtt{C}' = \mathtt{C}$ is lehet) és $s \wedge \mathtt{c}$ konzisztens marad. Tehát egy összetett korlát *erősíti* a korlát-tár tartalmát, ha a korlát ekvivalens egy egyszerű korlát és egy másik összetett korlát konjunkciójával. Ilyenkor az egyszerű korlát bekerül a korlát-tárba, az új összetett korlát pedig visszakerül a célsorozatba.

A korlát-erősítésnek két speciális esetét érdemes megjegyezni:

1. `C=C'`. Ilyenkor a `c`-re vonatkozó feltétel az, hogy $s \Rightarrow (\mathtt{C}\Rightarrow\mathtt{c})$, azaz a tárból és a `C` korlátból megpróbálunk egy egyszerű `c` korlátot kikövetkeztetni.
2. `C'=true`, azaz az s korlát-tár mellett `C`-nek létezik egy vele ekvivalens `c` párja, ahol `c` már egyszerű, így felvehetjük a korlát-tárba, `C'`-t pedig eldobhatjuk.

Egy példa korlát-erősítésre: a korlát-tár tartalma legyen $Y > 3$, az összetett korlát pedig $X > Y*Y$. Ekkor $X > Y * Y \wedge Y > 3 \Rightarrow X > 9$, ami már felvehető a korlát-tárba. Az $X > Y * Y$ korlátnak továbbra is „démonként” életben kell maradnia, hogy amikor a későbbiekben Y tartományára további szűkítéseket teszünk, akkor az egyúttal módosítani tudja X tartományát is.

A korlátmegoldó algoritmussal szemben támasztott követelmények

- *teljesség*: egyszerű korlátok konjunkciójáról mindig el tudja dönteni, hogy az konzisztens-e
- *inkrementalitás*: új korlát felvételekor ne bizonyítsa újra a teljes tár konzisztenciáját, csak azokat a korlátokat, amelyeket az új korlát felvétele érint
- *visszalépés támogatása*: a levezetés során ellentmondás esetén vissza tudja csinálni a korlátok felvételét
- *hatékonyság*

1.4. A CLP rendszerek felhasználási lehetőségei

- **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

1.5. Segédanyagok

Az SWI-Prolog hasonló szintaxisú `clpfd` függvénykönyvtárának dokumentációja: [2].

2. fejezet

CLP segédeszközök SICStusban

Ez a fejezet bemutatja azokat az eszközöket, amelyeket a SICStus Prolog kínál egy rá épülő CLP nyelv megvalósításához. Az eszközök megismerése után egy, a természetes számok tartományára épülő CLP nyelvet (CLP(MiniNat)) fogunk megvalósítani.

2.1. Korutinszervezés

A korutin egy olyan Prolog rutin, amely végrehajtása egy adott feltétel teljesüléséig felfüggeszthető. Amint a feltétel igazzá válik, a korutin újraaktiválódik és lefut. A feltétel legtöbbször bizonyos változók behelyettesítettségére vonatkozik, de a SICStus támogat más feltételtípusokat is. A korutinszervezés hatékonyabbá és átláthatóbbá teszi a programkódot, ezért érdemes használni.

2.1.1. Blokk-deklarációk

Lehetőség van arra, hogy előírjuk, hogy egy adott eljárás addig ne fusson le, amíg bizonyos paraméterváltozói be nem helyettesítődnek. Például:

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

Jelentése: ha a `p` hívás első és harmadik argumentuma is behelyettesítetlen, akkor a hívás függesztődjön fel. A hívás csak akkor folytatódik, ha az első vagy a harmadik argumentum nem-változó értéket kap. Ha a futás végén maradtak felfüggesztett hívások, akkor azokat a Prolog rendszer kiírja. Lehetőség van vagylagos blokkolási feltétel megadására is:

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

Jelentése: a `p` hívás csak akkor futhat le, ha az első *és* a második argumentum is behelyettesítődik, tehát ha az első *vagy* a második argumentum behelyettesítetlen, akkor a hívás felfüggesztődik.

1. példa: biztonságos `append/3` hívás megvalósítása

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

2. példa: többirányú összeadás

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

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

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

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

2.1.2. Blokkolás alkalmazása: végtelen választási pontok kiküszöbölése

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

pick([X|L], X, L).
pick([Y|L], X, [Y|L1]) :-
    pick(L, X, L1).

perm([], []).
perm(L, [X|P]) :-
   pick(L, X, L1), perm(L1, P).
```

A `perm` eljárás a egy adott lista permutációját állítja elő. Normál esetben (blokkolás nélkül) az első paramétere a bemenő, a második a kimenő. A matematikai érzék azt sugallja, hogy logikus lenne, ha bármelyik argumentum lehetne a bemenő (a permutáció kölcsönös). A `perm` eljárás blokk-deklarációval kiegészítve visszafelé is működik:

```
| ?- perm(L, [1,2]).
   1  1  Call: perm(_69,[1,2]) ?
   -  -  Block: pick(_363,1,_368)
   2  2  Call: perm(_368,[2]) ?
   -  -  Block: pick(_742,2,_747)
   3  3  Call: perm(_747,[]) ?
   -  -  Unblock: pick(_742,2,[])
   4  4  Call: pick(_742,2,[]) ?
   -  -  Unblock: pick(_363,1,[2])
   5  5  Call: pick(_363,1,[2]) ?
   5  5  Exit: pick([1,2],1,[2]) ?
   4  4  Exit: pick([2],2,[]) ?
   3  3  Exit: perm([],[]) ?
   2  2  Exit: perm([2],[2]) ?
   1  1  Exit: perm([1,2],[1,2]) ?
L = [1,2] ?
```

2.1.3. Blokkolás alkalmazása: generál-és-ellenőriz típusú programok gyorsítása

A generál-és-ellenőriz típusú programok valamilyen módszerrel generálják a lehetséges megoldásokat, és ezután ellenőrzik, hogy az aktuálisan vizsgált lehetőség jó-e. Ezek a programok általában nem hatékonyak, mert túl sok visszalépést használnak. Korutinszervezéssel a generáló és ellenőrző rész „automatikusan” összefésülhető, így a végrehajtás sokkal hatékonyabbá tehető. Ehhez az ellenőrző részt előre kell tenni és megfelelően blokkolni. Az alábbi példa egy buta rendező algoritmust javít fel viszonylag elfogadható sebességűre. A rendezés alapja: generáljuk a rendezendő lista összes permutációját, majd ellenőrizzük, hogy rendezett-e.

```
% az egyszerű generál-és-ellenőriz típusú programhoz
% a sorted és a perm felcserélendő
sort(L, S) :- sorted(S), perm(L, S).

sorted([]).
sorted([\_\,]).
sorted([X,Y|L]):- sorted(L, X, Y).

:- block sorted(?, -, ?), sorted(?, ?, -).
sorted([], X, Y) :- X =< Y.
sorted([Z|L], X, Y) :- X =< Y, sorted(L, Y, Z).
```

Futási idők

Listahossz	7	8	9
gen-test	0.48s	3.86s	35.64s
korutinos	0.04s	0.09s	0.18s
gen-test+korutin	0.55s	4.37s	40.71s

Megjegyzés: a táblázat utolsó sora azt jelzi, hogy a blokkolásért árat kell fizetni: ha feleslegesen alkalmazzuk, lelassíthatja a programot.

2.1.4. További korutinszervező eljárások

Hívások késleltetésére a `freeze/2`, `dif/2` és `when/2` eljárások is felhasználhatóak. A `freeze(X,Hívás)` mindaddig felfüggeszti a megadott hívást, amíg `X` behelyettesítetlen változó. Mivel a hívás a Prolog `call/1` eljárásával hajtódik végre, és ez elég nagy overhead-del rendelkezik, célszerű a `freeze` `block`-kal való helyettesítése, ahol csak lehet. A `freeze(X,Hívás)` egy lehetséges megvalósítása:

```
:- block freeze(-,?).
freeze(_,Hivas) :- call(Hivas).
```

A `dif(X,Y)` egy olyan cél, amely akkor sikerül, ha `X` és `Y` nem egyesíthető, de mindaddig felfüggeszti a végrehajtását, amíg ez el nem dönthető. Az általános felfüggesztés megvalósítására a `when(Feltétel, Hívás)` eljárás használható. Ez mindaddig felfüggeszti `Hívást`, amíg `Feltétel` nem teljesül. A `Feltétel` egy nagyon leegyszerűsített Prolog cél lehet, amely szintaxisa:

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

Ebben a fenti szintaxisban a `nonvar(X)` jelentése: X nem változó. A `ground(X)` feltétel azt várja el, hogy `X` tömör legyen, azaz ne tartalmazzon behelyettesítetlen változót. `?=(X,Y)` jelentése: `X` és `Y` egyesíthetősége eldönthető. A vesszővel elválasztott feltételek konjunkcióba, a pontosvesszővel elválasztottak diszjunkcióba kerülnek egymással. Egy egyszerű példa a `when/2` használatára:

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

A fenti példában a `process(X,Y,T)` cél akkor fut le, ha `T` nem tartalmaz behelyettesítetlen változót, és vagy `X` nem változó, vagy pedig `X` és `Y` egyesíthetősége eldönthető.

A késleltetett hívások lekérdezésére a `frozen/2` és a `call_residue/2` eljárások használhatóak. A `frozen(X,Hívás)` meghatározza az `X` változó miatt felfüggesztett hívásokat, és azokat egyesíti `Hívás` értékével. A `call_residue(Hívás,Maradék)` végrehajtja `Hívást`, a végrehajtás után felfüggesztve maradt eljárásokat pedig `Maradék`ban adja vissza. Például:

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

Látható, hogy a `Maradek` változó egyúttal feltünteti azt is, hogy melyik hívás melyik változók miatt maradt felfüggesztve.

2.2. További Prolog eszközök a CLP nyelvek megvalósítására

Tetszőleges nagyságú egész számok

A Prologban tetszőleges nagyságú egész számokat tárolhatunk, nincs rájuk korlát, mint a legtöbb programozási nyelvben. Például ha írtunk egy `fakt/2` eljárást, amely minden n egész számra kiszámítja $n!$-t, akkor semmi akadálya annak, hogy nagy n-ekre is lefuttassuk az eljárást:

```
| ?- fakt(100,F).
F = 93326215443944152681699238856266700490715968264381
62146859296389521759999322991560894146397615651828625 3
69792082722375825118521091686400000000000000000000000 ?
```

Visszaléptethető módon változtatható kifejezések (mutábilisek)

Ezek tulajdonképpen a gépközelibb programozási nyelvek pointer fogalmát hozzák be a Prolog világba. Ha például építünk egy Prolog fastruktúrát, aminek két (vagy több) részfájában ugyanarra a változtatható kifejezésre van szükségünk, akkor a mutábilisek használatával ha az egyik helyen megváltoztatjuk a kifejezést, akkor a másik helyen is megváltozik. Mutábilisek használata nélkül ezt nehéz és nem is hatékony megírni, főleg ha a visszalépést is figyelembe kell venni.

- `create_mutable(Adat, Kif)`
  `Adat` kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz `Kif`. `Adat` nem lehet üres változó.
- `get_mutable(Adat, Kif)`
  `Adat`-ba elővesz `Kif` pillanatnyi értékét.
- `update_mutable(Adat, Kif)`
  `Adat`-ra változtatja `Kif` értékét. A változtatás visszalépéskor visszacsinálódik. `Adat` nem lehet üres változó.

Mellékhatás visszavonása visszalépéskor

Mellékhatásos eljárások esetén lehetőség van a mellékhatások visszavonására, ha visszalépés történik. A mellékhatásokat visszavonó eljárást egy `undo/1` hívásba kell ágyazni, és beleírni a Prolog kódba. Az `undo(Kif)` feltétel és mellékhatás nélkül mindig sikerül, de ha visszalépés történik, akkor végrehajtja `Kif`-et. Például:

```
assert_b(Cl) :- assert(Cl), undo(retract(Cl)).
```

2.3. A CLP(MiniNat) nyelv megvalósítása

A CLP(MiniNat) nyelv jellemzése

- Tartomány ($\mathcal{D}$): a nem negatív egészek halmaza
- Függvények ($\mathcal{F}$): összeadás, kivonás, szorzás
- Korlát relációk ($\mathcal{R}$): $=, <, >, \leq, \geq$
- Korlát-megoldó algoritmus ($\mathcal{S}$): a SICStus korutin-kiterjesztésén alapul
- A Prolog-ba ágyazás szintaxisa:
  `{`*Korlát*`}` jelenti egy adott korlát felvételét. A `{…}` konstrukció csak szintaktikai édesítőszer, valójában a `'{}'/1` struktúrát takarja

Példafutás

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

2.3.1. Számábrázolás

A korábban látott `plusz/3` eljárásban (ld. 2.1.1 fejezet) az N szám ábrázolására egy N elemű listát használtunk. A lista elemei érdektelenek voltak, ezért behelyettesítetlen változóval ábrázoltuk őket. Például a 3-as szám ábrázolása így nézett ki: `[_,_,_]` ≡ `.(_,.(_,.(_,[])))`. Ha elhagyjuk a behelyettesítetlen változókat, és a `./2` helyett az `s/1` struktúrát, valamint a `[]` konstans helyett a `0` számot használjuk, akkor a fenti példában az alábbi alakhoz jutunk: `s(s(s(0)))`. Itt az `s` az angol *successor* (*követő*) szó rövidítése, és ez jól kifejezi a lényeget: ebben az úgynevezett Peano-féle számábrázolási módban mindent a 0 konstanssal és az `s` operátorral fejezünk ki, ahol `s(X)` az X szám követőjét jelenti. Ezt a számábrázolást fogjuk felhasználni az általunk megvalósítandó CLP(MiniNat)-ban, tehát `0=0`, `1=s(0)`, `2=s(s(0))` stb.

2.3.2. Összeadás és kivonás megvalósítása

Az előző fejezetben vázolt számábrázolási mód segítségével az összeadás és a kivonás megvalósítása:

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

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

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

A `plusz/3` predikátum itt elvárja, hogy a két bemenő és egy kimenő paraméter közül vagy az első bemenő, vagy a kimenő behelyettesített legyen. Mivel az összeadásnál a tagok sorrendje indifferens, ezért ezt a megkülönböztetést (az első és a második bemenő paraméter megkülönböztetését) valahogy el kell fednünk. Erre szolgál a `+/3` predikátum, amelyik már akkor lefut, ha a három argumentuma közül bármelyik behelyettesített, és ha történetesen az első argumentum pont behelyettesítetlen lenne, akkor megcseréli az első kettőt és úgy hívja meg a `plusz/3` predikátumot.

2.3.3. A szorzás megvalósítása

A szorzás megvalósítása során az alábbi alapelvekhez tartjuk magunkat:

- Mindaddig felfüggesztve tartjuk a célt, amíg legalább az egyik tényező vagy a szorzat be nem helyettesítődik.
- Ha az egyik tényező behelyettesített, akkor a célt ismételt összeadásra vezetjük vissza.
- Ha a szorzat behelyettesített, akkor az egyik tag helyére rendre behelyettesítjük 1-et, 2-t, ..., N-et (ahol N a szorzat), majd mindegyik lehetőséget ismételt összeadásra vezetjük vissza.

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

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

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

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

2.3.4. Korlátok lefordítása célsorozatokra

Az előző két fejezetben megvalósítottuk az összeadás, kivonás, szorzás műveleteket Prolog eljárások formájában. Szükségünk lesz azonban egy olyan eljárásra is, amely a „hagyományos” matematikai kifejezések formájában leírt korlátokat lefordítja ezekre az eljárásokra, majd az ily módon összeállított célsorozatot meghívja. Például az `X*Y+2=Z` korlát lefordított alakja: `*(X,Y,_A), +(_A,s(s(0)),Z)`.

Egyúttal a fenti eljárás vissza is fogja vezetni a `=<, <, >=, >` korlátokat a már megvalósított korlátokra: az `X =< Y`-t az `X+_=Y`, az `X < Y`-t pedig az `X+s(_)=Y` hívás fogja helyettesíteni.

Először felveszünk egy eljárást, amely lehetővé teszi, hogy a korlátokat `{Korlát}` alakú kifejezésekkel vehessük fel:

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

A korlátok fordításához három eljárásra lesz szükségünk:

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

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

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

Amint látható, egy `Kif1` *Op* `Kif2` kifejezés lefordított alakja egy három részből álló célsorozat, amely egy E változóban állítja elő a kimenetét. A célsorozat első eleme meghatározza a `Kif1` értékét E_1-ben előállító célsorozatot, a második eleme meghatározza a `Kif2` értékét E_2-ben előállító célsorozatot, végül a harmadik eleme az `Op(E1,E2,E)` hívás, ahol `Op` a `+`, `-`, `*` jelek egyike. Ha egy kifejezés helyén csak egy szám áll önmagában, akkor az ő lefordított formája az ő Peano-alakja. Minden egyéb (változó, vagy Peano-alakú szám) változatlan formában marad a fordításkor.

2.3.5. Formázott kiírás

Természetes elvárás a rendszerrel szemben, hogy az esetlegesen (pl. nyomkövetés esetén) kiírásra kerülő Peano-számokat ne a CLP(MiniNat) belső ábrázolási formájában, hanem a hagyományos formátumban írja ki a képernyőre. Ennek megvalósításában segít a `print/1` és a `portray/1` eljárás.

A `print/1` alapértelmezésben megegyezik a `write/1` Prolog kiíró eljárással. Ha azonban definiálva van a `portray/1` „kampó” eljárás (*hook predicate*), akkor először minden kiírandóra meghívja `portray`-t, és ha ez a hívás meghiúsul, akkor maga írja ki a paraméterként átadott struktúrát. A `print/1` eljárást használja a Prolog rendszer többek között a változó-behelyettesítések és a nyomkövetési kimenet kiírására is, így ha definiálunk egy megfelelő `portray/1` eljárást a Peano-számok formázására, akkor ezzel el is értük az első bekezdésben felvázolt célt. Hasonló módon felvehetünk még egy `portray` predikátumot a felfüggesztett célok kiírásának formázására is.

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

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

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

2.3.6. Klózok fordítási időben történő átalakítása

Az eddig összeállított CLP(MiniNat) rendszerünk már használható, azonban teljesítményét jelentősen rontja, hogy a `'{}'/1` struktúrával felvett korlátokat csak futási időben alakítja át célsorozatokra. Lehetőség van arra is, hogy a betöltött programon még a futtatás előtt, fordítási időben hajtsunk végre bizonyos változtatásokat, transzformációkat, például egy ilyen jellegű átalakítást. Ezeket a műveleteket a `term_expansion/2` és `goal_expansion/3` eljárásokkal valósíthatjuk meg.

- `term_expansion(+Kif, -Klózok)`
  Minden betöltő eljárás (`consult`, `compile` stb.) által betöltött kifejezésre a rendszer meghívja. A kifejezést a `Kif` paraméterben adja át, a transzformált alakot a `Klózok` paraméterben várja

(ez akár lista is lehet). Ha az eljárás meghiúsul, akkor a rendszer a kifejezést változatlan alakban veszi fel.

- `goal_expansion(+Cél, +Modul, -ÚjCél)`
  Minden, a programból vagy a szabványos bemenetről beolvasott részcélra meghívja a rendszer. A transzformált célt az `ÚjCél` paraméterben várja. Ha az eljárás meghiúsul, akkor a rendszer a célt változatlan alakban használja fel.

A `goal_expansion/2` használatával a korlátok fordítási idejű átalakítása a következőképpen írható le:

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

Érdemes összehasonlítani egy egyszerű faktoriálisszámító CLP(MiniNat) program korlátokkal leírt és lefordított változatát:

```
:- block fact(-, -).            :- block fact(-, -).

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

Amint látható, a második példa már nem foglalkozik a számok Peano-alakra hozásával, azt nekünk kell külön elvégezni:

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

2.3.7. További problémák a CLP(MiniNat)-ban

Kis kísérletezés után könnyen rátalálhatunk a CLP(MiniNat) alábbi problémájára (amit a nulla szorzat problémájának nevezhetünk):

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

A Prolog programokban a kétszeresen adódó megoldások általában nemkívánatosak. A probléma kiküszöböléséhez kicsit módosítanunk kell a `szorzatuk_nulla/2` eljárásunkat:

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

Amint az alábbi példák mutatják, ez a kezdeti problémánkat megoldja, de még mindig nem tökéletes, ugyanis ha X és Y egyesíthetősége a korlát hívása után dönthető csak el, akkor a kettőzött megoldás ugyanúgy előadódik:

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

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

A végleges javításhoz fel kell használnunk a `dif/2` eljárást, amely felfüggeszti a `szorzatuk_nulla/2` eljárás második ágát addig, amíg az egyesíthetőség el nem dönthető:

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

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

A másik problémát erőforrás problémának hívjuk. Ez a rekurzív `fact/2` eljárás használata esetén adódik. Tekintsük például a `fact(X,11)` hívást, amely megkeresné azt az `X` számot, melyre `X!=11`. A hívást a második `fact` klózzal illesztve a `{11=X*F1}` hívásra tudjuk visszavezetni, ez pedig két megoldást generál (`X=1, F1=11` és `X=11, F1=1`). Ezekre a behelyettesítésekre feléled a rekurzív `fact` hívás `fact(0,11)` és `fact(10,1)` paraméterekkel. Az első hívás azonnal meghiúsul, a másodikhoz viszont a Prolog „mohó” módon megpróbálja kiszámolni 10!-t, és csak utána egyesítené az eredményt 1-gyel, 10! azonban Peano-szám formában nem ábrázolható, mert nincs hozzá elég memória. A probléma úgy javítható, hogy a szorzat-feltétel felvételét még a rekurzív hívás elé kell tenni a `fact/2` eljárás második klózában:

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

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

Általános szabályként megállapíthatjuk, hogy egy korlát-programban célszerű minél kevesebb választási pontot csinálni, és éppen ezért az összes korlátot érdemes a tényleges keresés *előtt* felvenni. A legtöbbször a keresésre egy úgynevezett *címkéző* (*labeling*) eljárást használunk, amely szisztematikusan, valamilyen módszer szerint végigpróbálgatja a nem lekötött változók lehetséges értékeit. CLP(MiniNat)-ban egy ilyen eljárás megvalósítása nehézkes, ezért itt nem foglalkozunk vele. CLP(MiniB)-ben (a Boole értékek halmazán dolgozó CLP megvalósításban) viszont könnyű: minden változóra a 0 és az 1 értéket kell kipróbálnunk.

3. fejezet

A SICStus `clpq` és `clpr` könyvtárai

A következő fejezetben a SICStus `clpq` , illetve `clpr` könyvtáraival fogunk foglalkozni.

3.1. A `clpq` és `clpr` könyvtár általános jellemzése

A `clpq` könyvtár egy, a racionális számok tartományára alapuló CLP rendszert valósít meg, a `clpr` könyvtár pedig ugyanezt, csak lebegőpontos formában ábrázolt valós számokkal. A felhasználható függvények tartalmazzák az alapműveleteket (`+ - * /`), valamint több magasabb rendű műveletet is (`min`, `max`, `exp`, `abs`, `sin`, `tan`...). Korlát-relációnak a valós számok között fennálló alapvető relációkat használhatjuk (`= =:= < > =< >= =\=`). Egyszerű korlátoknak a lineáris összefüggéseket tartalmazó korlátokat tekintjük, a korlátmegoldó algoritmus a Gauss-elimináción és a szimplex módszeren alapul. A könyvtárakat az alábbi parancsokkal vehetjük használatba:

```
:- use_module(library(clpq)).
:- use_module(library(clpr)).
```

A korlát-tárat a CLP(MiniNat)-hoz hasonlóan a `{Korlát}` alakú kifejezésekkel bővíthetjük, ahol `Korlát` egy változókból és egész vagy lebegőpontos számokból a fenti műveletekkel felépített reláció, vagy ilyen relációk vesszővel elválasztott konjunkciója.

Mivel a `clpq` és a `clpr` könyvtárak nagy mértékben hasonlítanak egymásra, ezért a továbbiakban csak a `clpq`-val foglalkozunk, de az elmondottak ugyanúgy érvényesek a `clpr`-re is.

3.2. Egy példafutás a `clpq` könyvtár segítségével

Az alábbiakban egy rövid példán keresztül fogjuk bemutatni a `clpq` könyvtár használatát.

Először be kell töltenünk a `clpq` könyvtárat, hogy használatba vehessük:

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

Egyszerű korlátok felvétele (lineáris egyenletrendszer megoldása):

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

Ha a beadott korlátoknak még nincs egyértelmű megoldása, akkor a `clpq` rendszer a fennálló relációkat (a korlát-tár állapotát) írja ki, mint például az alábbi lineáris egyenlőtlenségnél:

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

Mint már említettük, a `clpq` rendszer csak lineáris korlátokat tud felvenni a korlát-tárba, előfordulhat azonban, hogy egy nemlineáris korlátot linearizálni tud. Ilyen esetben a nemlineáris korlát lineáris megfelelője be tud kerülni a korlát-tárba. Az alábbi példa egy olyan esetet mutat, amikor egy kifejezés két különböző, de ekvivalens alakját beadva az egyik a linearizálás miatt be tud kerülni a korlát-tárba, míg a másik nem:

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

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

Tisztán nemlineáris korlátok minden esetben a táron kívül maradnak, persze előfordulhat, hogy a nemlineáris korlát egy későbbi egyesítés vagy változóbehelyettesítés miatt lineárissá válik, mint ahogy az alábbi példa is mutatja:

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

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

Persze a nemlineáris korlátok is megoldhatóak:

```
| ?- {2 = exp(8, X)}.
X = 1/3 ?
```

3.3. Összetett korlátok kezelése `clpq`-ban

Ahogy azt már említettük, `clpq`-ban és `clpr`-ben összetett korlátnak számítanak a nemlineáris (vagy a rendszer által nem linearizálható, de egyébként lineáris) kifejezések, ezek nem kerülnek be a korlát-tárba, hanem démonként várakoznak arra, hogy lineárissá válva bekerülhessenek oda. Fontos megjegyezni, hogy a `clpq`-ban és a `clpr`-ben a démonok *semmiféle* erősítő tevékenységet nem végeznek, kizárólag akkor módosíthatják a tárat, ha valamilyen behelyettesítés folyamán lineárissá válnak. Ez gyengébb az általános korláterősítő mechanizmusnál. Ennek bizonyítására képzeljük el a következő példát: a korlát-tár tartalma legyen az `X > 3` korlát, a nemlineáris korlát pedig az `Y > X*X`. A korlátmegoldó ennek alapján kikövetkeztethetné, hogy `Y > 9`, és ezt fel is vehetné a korlát-tárba, hiszen ez már lineáris korlát (persze emellett az `Y > X*X` démont továbbra is életben kell hagynia). A `clpq/r` ezt nem teszi meg, és így nyilvánvalóan nem használja fel ezt az információt a további következtetésekhez.

Lássunk egy további példát az összetett korlátokra vonatkozóan!

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

Nézzük meg, hogyan jött ki a fenti eredmény! A rendszer a fenti célsorozat futtatásakor választási pont létrehozása nélkül felveszi az első két korlátot a korlát-tárba, ezt az alábbi célsorozat futtatásával ellenőrizhetjük:

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

A sor végén jól látható a várakozó démon is. Ezek után egy választási pont következik, és az első ágon továbbhaladva felvesszük a `Z`-re vonatkozó korlátot:

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

Látható, hogy a démonunk felébredt, és egyszerű korláttá válva bekerült a korlát-tárba. Ha ezek után megérkezik az `Y`-ra vonatkozó korlát, akkor ez ellentmondásban lesz a korlát-tár eddigi tartalmával, hiszen ha `X` pozitív (lásd az előző futás eredményében az utolsó korlátot), és `Y` negatív, akkor `X-Y` mindenképp pozitív, ami ellentmondásban van azzal a korláttal, hogy `X-Y=<0`. Ezt a rendszer egy meghiúsulás formájában „éli át”:

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

A meghiúsulás miatt a korlát-tár tartalma visszafejtődik egészen a választási pontnál fennálló helyzetig, ahonnan a másik ágon fut tovább, és ott meg is találjuk az egyetlen megoldást:

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

3.4. Bonyolultabb `clpq` példa: hiteltörlesztés

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

A fenti `clpq` példa további magyarázatot nem igényel, érdemes azonban megfigyelnünk, hogy a `clpq` természetéből adódóan az eljárás hívásakor nem kötelező úgy kitöltenünk a paramétereket, hogy azokból egyértelműen következzen a megoldás, ugyanis ha ez nem teljesül, akkor a `clpq` kifejezi a hiányzó adatokat a megadottak függvényében:

```
| ?- 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-ot törleszt havonta,
                        % hány hónapig kell törleszteni?
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} ?
```

Az `ordering/1` predikátum egy listát vár paraméterként, és ezzel a listával megadhatjuk, hogy az eredményben milyen sorrendben szerepeljenek a változók. Ezzel gyakorlatilag azt is szabályozhatjuk, hogy melyik változót melyikek segítségével fejezze ki a rendszer. A fenti példában `P` áll a lista első helyén, ezért `P`-t fogja kifejezni a többi segítségével.

3.5. További könyvtári eljárások

- `entailed(Korlát)` — sikerül, ha `Korlát` levezethető a jelenlegi tárból, meghiúsul, ha nem
- `inf(Kif,Inf)`, `sup(Kif,Sup)` — kiszámolja `Kif` infimumát, illetve szuprémumát, és egyesíti `Inf`-fel, illetve `Sup`-pal. Példa:

  ```
  | ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y }, sup(Z, Sup).
  Sup = 310, {....}
  ```

- `minimize(Kif)`, `maximize(Kif)` — kiszámolja `Kif` infimumát, illetve szuprémumát, és egyesíti `Kif`-fel. Példa:

  ```
  | ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y }, maximize(Z).
  X = 7, Y = 2, Z = 310 ?
  ```

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

```
| ?- {X >= 0.5, Y >= 0.5}, inf(X+Y, I).
I = 1, {Y>=1/2}, {X>=1/2} ?

| ?- {X >= 0.5, Y >= 0.5}, bb_inf([X,Y], X+Y, I).
I = 2, {X>=1/2}, {Y>=1/2} ?
```

- `ordering(V1 < V2)` — A `V1` változó előbb szerepeljen az eredmény-korlátban mint a `V2` változó.
- `ordering([V1,V2,...])` — `V1, V2, ...` ebben a sorrendben szerepeljen az eredmény-korlátban.

3.6. A `clpq` és a `clpr` belső számábrázolása

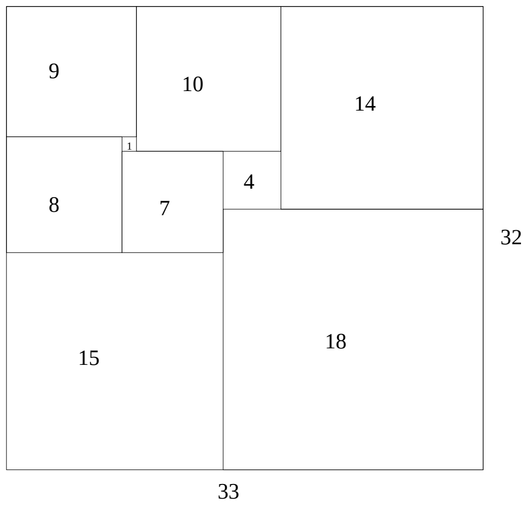
A `clpr` lebegőpontos szám formátumban tárolja a számokat, itt tehát semmi különlegességgel nem találkozhatunk. A `clpq` azonban racionális számokkal dolgozik, és ezeket a számokat egy `rat(Számláló,Nevező)` alakú struktúrával ábrázolja, ahol a tört számlálója és nevezője mindig relatív prím. Ennek bizonyítására tekintsük az alábbi `clpq` példákat:

```
| ?- {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 ?
```

3.7. Egy nagyobb `clpq` feladat: tökéletes téglalapok

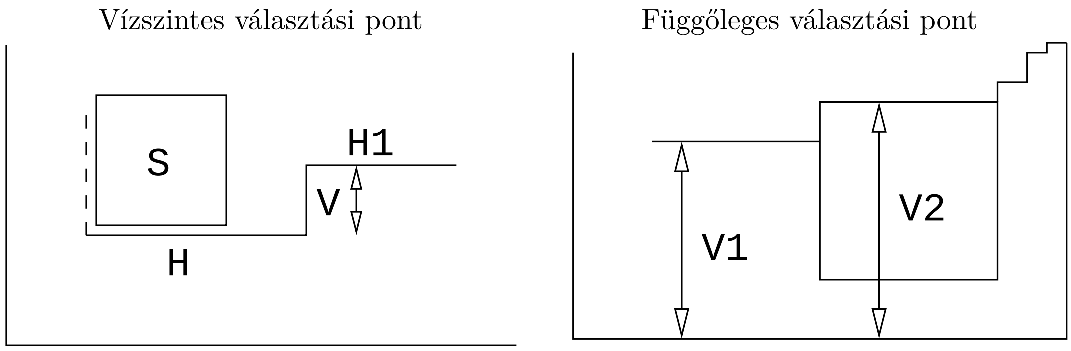
A feladat: egy olyan téglalap keresése, amely kirakható páronként különböző oldalú négyzetekből.

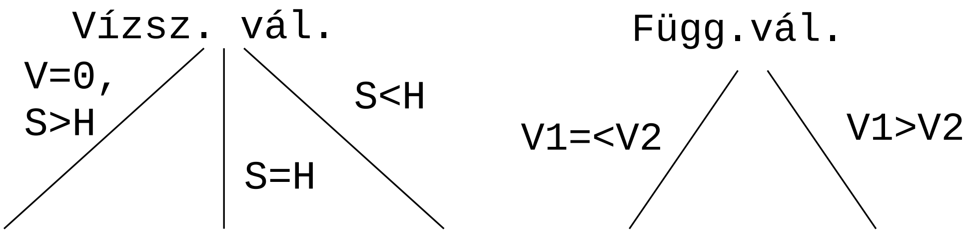
Egy lehetséges megoldás
(a legkevesebb, 9 db négyzet felhasználásával)



A feladat megoldása során a négyzetet a „tetris-elv” alapján, alulról felfelé és balról jobbra fogjuk kitölteni, tehát a következő négyzetet mindig igyekszünk a négyzetben lent és bal oldalt elhelyezni, ameddig ez lehetséges. Mivel a fenti elv alapján töltjük ki a négyzetet, ezért a ki nem töltött terület összefüggő, ezért jellemezhető a körvonalával, amit Prologban egy listával fogunk lekódolni, amelyben a vonal függőleges és vízszintes szakaszainak hosszát adjuk meg egy adott körüljárási sorrend szerint. A függőleges és vízszintes szakaszok váltakozva fordulnak elő, ezért a listánkban minden páratlanadik elem függőleges szakaszt, minden párosadik elem vízszintes szakaszt fog kódolni. A körüljárást a négyzet bal felső sarkából kezdjük az óramutató járásával ellenkező irányban, és a lista utolsó elemét elhagyjuk, mivel a körvonal záródása miatt ez úgyis redundáns. Ezek alapján például egy 33×32-es üres négyzet a `[-32,33,32]` listával kódolható. Ha tekintjük a fenti négyzetet abban az állapotban, amikor csak a 15 oldalhosszúságú négyzetet helyeztük el, akkor az üres területet a `[-17,15,-15,18,32]` lista írja le.

A keresési terünkben kétfajta választási pont fog előfordulni: vízszintes és függőleges. Függőleges választási pontnál azt döntjük el, hogy amikor egy négyzet mellé egy másik négyzetet lerakunk, akkor az új négyzet magassága milyen relációban álljon a már lerakott négyzettel. Vízszintes választási pontnál azt döntjük el, hogy a már kiválasztott oldalhosszúságú négyzet mellett mekkora üres helyet hagyjunk még a kitöltésben. Ezt a két választási pontot szemlélteti az alábbi ábra:





A fentiekhez még annyi kiegészítés szükséges, hogy a programban a négyzet függőleges oldalát mindig 1-nek választjuk, így csak a vízszintes oldal méretével kell variálni. A megoldáshoz használt predikátumok:

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

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

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

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

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

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

A program belépési pontja a `filled_rectangle/2` predikátum, amely a `Width` paraméterében fogja megadni a téglalap szélességét, `Ss`-ben pedig a kitöltéshez felhasznált négyzetek listáját. A `distinct_squares/1` korlát adja meg, hogy a négyzeteknek nem lehetnek azonos méretűek. Ez egy egyszerű „darálás" az `Ss` listában előforduló összes négyzet-párra az `outof/2` segítségével. A négyzetek elhelyezését a `filled_hole/4` predikátum végzi. A predikátum harmadik és negyedik paramétere a kezdő- és a végállapotban elhelyezett négyzetek listáját adja, az első és a második paraméter pedig a harmadik és negyedik paraméterben leírt állapothoz tartozó határoló vonalak leírása a fentebb leírt formában. A `placed_square/3` predikátum pedig a vízszintes választási pontok három fajtáját írja le, mégpedig azt, hogy ilyen esetben a határoló vonal milyen szabályok szerint változik.

Lássunk egy példafuttatást:

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

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

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

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

Amint látható, 9-nél kevesebb négyzettel nem lehet lefedni a téglalapot, 9-re viszont máris három megoldást talált a program, ebből az első és a harmadik csak a négyzetek elhelyezésében különbözik.

A program működésének megértéséhez hagyjuk ki az `outof/2` által generált korlátokat, és nézzük meg, hogy kisebb méretű téglalapokra milyen korlátokat generál a program! Kommentként közöljük az adott ágon generált korlátokat és lefedést, a rendundáns korlátok elhagyásával. A `filled_rectangle/3` `[eqsq]` paramétere jelzi, hogy most megengedünk azonos méretű négyzeteket.

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

S1 = 1/2, S2 = 1, S3 = 1/2, W = 3/2 ? ;     % 3 3 2 2 2 2
```

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

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

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

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

4. fejezet

A SICStus `clpb` könyvtára

A következő fejezetben a SICStus `clpb` könyvtárával fogunk foglalkozni. A `clpb` könyvtárat az alábbi módon lehet használatba venni:

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

4.1. A `clpb` könyvtár általános jellemzése

A `clpb` könyvtár a kétértékű Boole-logikán alapuló CLP rendszert valósít meg, ennek megfelelően a `clpb` változók értékkészlete a $0, 1$ halmaz.

Felhasználható függvények (egyben korlát-relációk)

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

A `clpb`-ben az összes korlát egyszerű korlát, nincsenek összetett korlátok. Pont ez a tény az, ami a `clpb`-t nagy feladatok megoldására alkalmatlanná teszi, hiszen minden korlát azonnal bekerül a korlát-tárba, és egy idő után a megoldó algoritmus (a Boole-egyesítés) működése a korlát-tár nagy mérete miatt lelassul.

Alapvető könyvtári eljárások

- `sat(Kifejezés)` – hozzáveszi `Kifejezés`t a korlát-tárhoz. `Kifejezés` a `0` és `1` konstansokból, atomokból, valamint változókból a fenti műveletekkel felépített logikai kifejezés. A kifejezésben előforduló atomok a kifejezés legkülső szintjén univerzálisan kvantifikált változókat jelentenek.

- `taut(Kifejezés,Érték)` – megvizsgálja, hogy `Kifejezés` levezethető-e a korlát-tárból. Ha levezethető, akkor `Értéket` 1-gyel egyesíti. Ha `Kifejezés` tagadása levezethető, akkor `Értéket` 0-val egyesíti. Minden más esetben meghiúsul.
- `labeling(Változók)` – beállítja a `Változók` lista összes elemét 1-re vagy 0-ra úgy, hogy a korlát-tár teljesüljön. Visszalépésre az összes lehetséges megoldást felsorolja.

4.2. Példafutások a clpb könyvtár segítségével

```
| ?- 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) ?
```

Látható, hogy a `clpb` a korlát-tár tartalmát `sat(Kifejezés)` alakú struktúrák konjunkciójaként jeleníti meg, ahol `Kifejezés` mindig egy „polinom”, azaz konjunkciók kizáró vagy (`#`) műveletekkel képzett sorozata. Az atomok a fent elmondottak szerint univerzálisan kvantifikált változókat jelentenek. Az univerzális és az egzisztenciális kvantifikáció különbségének kiemelésére nézzük meg az alábbi példákat is:

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

Nézzünk most egy picit komplikáltabb `clpb` példát, amely egy egy bites összeadó áramkör működését próbálja modellezni `clpb` korlátok segítségével:

```
| ?- [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
```

Az összeadó működése a `card/2` segítségével nagyon egyszerűen leírható: az összeg értéke akkor 1, ha az `[X,Y,Cin]` listában (`Cin` a carry in, tehát a bemenő átvitel rövidítése) 1 vagy 3 db egyes van, `Cout` (a kimenő átvitel) értéke pedig akkor 1, ha az `[X,Y,Cin]` listában legalább két db egyes van. Nézzük meg, hogy ezeket a korlátokat milyen formában tárolja a `clpb` rendszer!

```
| ?- adder(x, y, Sum, cin, Cout).
sat(Sum=:=cin#x#y),
sat(Cout=:=x*cin#x*y#y*cin) ?
```

Látható, hogy a `card/2` itt is konjunkciók kizáró vagy kapcsolatára vezetődött vissza. Mivel csak a `Sum` és `Cout` változók viselkedésére voltunk kíváncsiak, ezért a másik három változót egzisztenciálisan kvantifikáltuk. Nézzük meg azt az egyszerűsített esetet is, amikor `Cin`-t 0-ra kötjük, tehát nincs bemenő átvitel:

```
| ?- adder(x, y, Sum, 0, Cout).
sat(Sum=:=x#y),
sat(Cout=:=x*y) ?
```

Ha a megoldás nem egyértelmű, akkor a `labeling` eljárással tudjuk felsoroltatni az összes lehetséges megoldást:

```
| ?- 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
```

4.3. A Boole-egyesítés

A `clpb` könyvtár két Boole-kifejezés egyesítésére „meglepő módon” a Boole-egyesítés nevű algoritmust használja. A feladat pontos megfogalmazása: legyen adott a g és h Boole-kifejezés, és keressük a $g = h$ egyenletet megoldó legáltalánosabb egyesítőt (a továbbiakban ezt $mgu(g, h)$-val jelöljük). Mivel a $g = h$ egyenlet helyettesíthető a $g \oplus h = 0$ egyenlettel (ahol $\oplus$ a kizáró vagy műveletet jelenti, amit `clpb`-ben a `#` operátor jelöl), ezért a továbbiakban a Boole-egyesítés vizsgálatához elegendő az $f = 0$ alakú egyenletek megoldását vizsgálnunk. Az egyesítés minden lépése során egy $f = 0$-beli x formulaváltozót szeretnénk kifejezni a többi segítségével.

Legyen $f_x(1)$ az f-ből az $x = 1$ helyettesítéssel, az $f_x(0)$ pedig az f-ből az $x = 0$ helyettesítéssel kapott formula. $f = 0$ kielégíthetőségének szükséges feltétele $f_x(1) \wedge f_x(0) = 0$ kielégíthetősége. Fejezzük ki x-et $f_x(1)$ és $f_x(0)$ segítségével úgy, hogy $f = 0$ legyen!

$f_x(0)$	$f_x(1)$	x
0	0	bármi (w)
0	1	0
1	0	1
1	1	érdektelen

Ha x-et $x = (a \wedge \overline{w}) \oplus (b \wedge w)$ (Prolog: `X=A*~W # B*W`) alakban keressük, akkor a fentiek szerint a és b értéke az alábbiak szerint adódik:

$f_x(0)$	$f_x(1)$	x	a	b
0	0	w	0	1
0	1	0	0	0
1	0	1	1	1

A táblázat alapján az $a = f_x(0)$ és $b = \overline{f_x(1)}$ megfeleltetés tűnik a legegyszerűbbnek. Így alapján az egyesítési algoritmus működése az $f = 0$ alakú egyenlőségekre:

1. Ha f-ben nincs változó, akkor f-nek azonosan 0-nak kell lennie, egyébként ugrás a következő pontra.
2. Helyettesítsünk f-ben egy tetszőleges x változót az $x = (f_x(0) \wedge \overline{w}) \oplus (\overline{f_x(1)} \wedge w)$ kifejezéssel (Prolog: `X=`$f_x(0)$`*~W # ~`$f_x(1)$`*W`).
3. Folytassuk az egyesítést az $f_x(1) \wedge f_x(0) = 0$ egyenlőségre

Példák a Boole-egyesítésre:

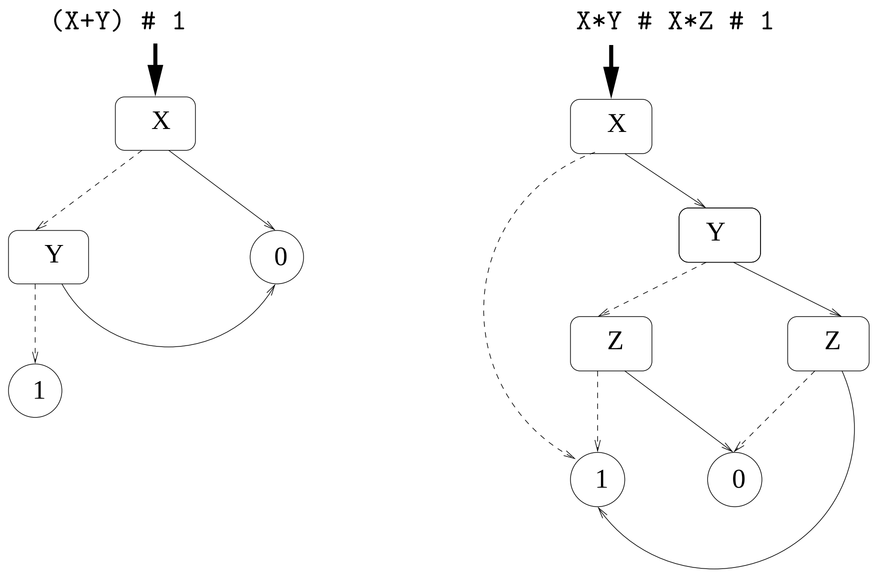
- mgu(`X+Y, 0`) $\longrightarrow$ `X = 0, Y = 0;`
- mgu(`X+Y, 1`) $=$ mgu(`~(X+Y), 0`) $\longrightarrow$ `X = W * Y # Y # 1;`
- mgu(`X*Y, ~(X*Z)`) $=$ mgu(`(X*Y)#(X*Z)#1, 0`) $\longrightarrow$ `X = 1, Y = ~Z.`

4.4. A `clpb` belső ábrázolási formája

A `clpb` könyvtár a Boole-kifejezéseket az úgynevezett *Boole/bináris döntési diagram*ok (*Boole/Binary Decision Diagram*s, *BDD*) segítségével ábrázolja. Ezek irányított körmentes gráfok (*directed acyclic graph*, *DAG*), amelyekben kétféle csomópont és kétféle él szerepel. A csomópontok tartozhatnak változóhoz vagy a 0 és 1 konstansokhoz. Csak a változókat tartalmazó csomópontokból indulhat ki él, mégpedig mindkettőből kétfajta, az egyik a hamis értéknek (az ábrákon szaggatott vonal), a másik az igaz értéknek (az ábrákon folytonos vonal) felel meg. Az egyik csomópont kitüntetett abból a szempontból, hogy ebbe a csomópontba egy végpont nélküli él is befut (ezt a csomópontot hívjuk *kezdő csomópont*nak). A BDD-k segítségével meghatározható az általuk reprezentált Boole-kifejezés értéke. Ehhez a gráfot a kezdő csomópontból kezdve kell bejárni a következő szabályok szerint:

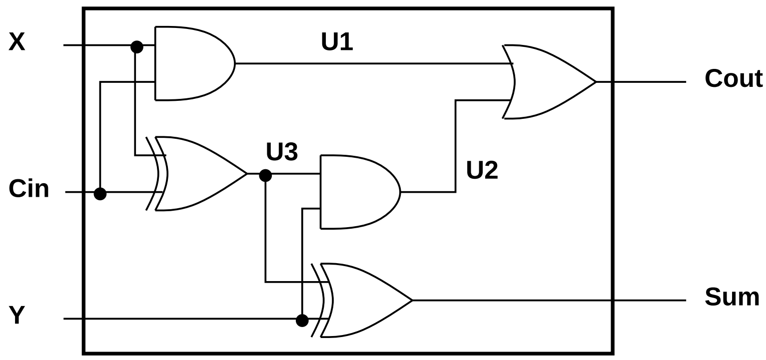
1. Ha az aktuális csomópont a 0 vagy az 1 konstansot tartalmazza, akkor megállhatunk a bejárással, és a kifejezés értéke 0 vagy 1, a csomópont tartalmától függően.
2. Ha az aktuális csomópont változót tartalmaz, akkor a változó aktuális értékétől függően vagy a szaggatott (0-nak megfelelő), vagy a folytonos (1-nek megfelelő) él mentén kell folytatni a bejárást.

Mivel a gráf irányított, körmentes, és csak a 0-t vagy 1-et tartalmazó csomópontok nyelők, ezért az algoritmus véges időn belül le fog futni. A könnyebb érthetőség kedvéért álljon itt két Boole-kifejezés bináris döntési diagramja:



4.5. Összetett `clpb` példa: hibakeresés áramkörben

Legyen adott egy egybites összeadó áramköri modellje (funkcionális elemekkel és összeköttetésekkel). Írjunk egy olyan `clpb` programot, amely adott bemenet-kimenet párra megmondja, hogy melyik funkcionális elem működik hibásan, ha feltételezzük, hogy egyszerre csak egy hibásodik meg!




```
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))
            ).
```

Amint látható, a `fault/3` predikátumban semmi mást nem tettünk, mint specifikáltuk az egyes áramköri elemek működését, valamint a `card` segítségével megadtuk azt a tényt is, hogy egyszerre legfeljebb egy áramköri elem hibás. Lássunk néhány példát a `fault` működésére!

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

Itt a `fault` helyesen kikövetkeztette, hogy ilyen bemenet-kimenet pár esetén csak a 4. funkcionális elem, azaz a programkód alapján az `X`-re és `Cin`-re kapcsolódó XOR kapu lehet hibás.

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

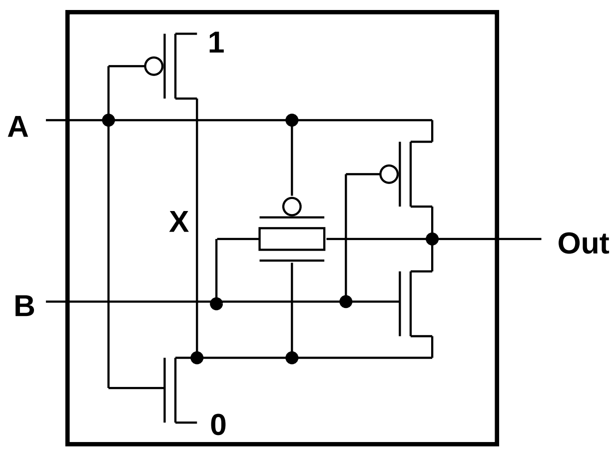
Itt a `fault` úgy gondolja, hogy vagy az első, vagy a harmadik funkcionális elem a hibás, ezt a korlát-tárban lévő `sat(_A=\=_B)` fejezi ki. Ha ehelyett látni szeretnénk a két lehetőséget teljesen behelyettesített változókkal, akkor szükséges a `labeling` használata is:

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

Végül megkérdezhetjük a `fault`-tól, hogy helyes áramköri működést feltételezve mik az egyes kimenetekhez rendelhető logikai egyenletek:

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

Tekintsünk most egy XOR kaput megvalósító tranzisztoros áramkört, és valósítsuk meg ennek a verifikációját `clpb` programmal!



Először leírjuk az npn és pnp tranzisztorok működését `clpb` kifejezésekkel:

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

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

Ezek segítségével megfogalmazhatjuk az XOR kapu működését, ha a földelést 0-val, a tápfeszültséget 1-gyel jelöljük:

```
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).
```

Ellenőrizzük, hogy az npn és pnp tranzisztorokra felírt egyenleteink helyesen működnek-e a gate feszültség ki- és bekapcsolására:

```
| ?- n(D, 1, S).                  S = D ?
| ?- n(D, 0, S).                  true ?
| ?- p(D, 0, S).                  S = D ?
| ?- p(D, 1, S).                  true ?
```

A fentiek szerint a tranzisztorok logikai felírása helyes, így nem maradt más hátra, mint hogy ellenőrizzük, hogy az XOR kapu ténylegesen az XOR függvényt valósítja-e meg:

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

4.6. Aknakereső játék clpb-ben

Az alábbi programkód egy aknakereső játékot valósít meg `clpb` programkód formájában.

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

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

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

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

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

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

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

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

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

A játék belépési pontja a `mine(Rows,Cols,Mines,Bd)` eljárás, amely egy `Rows` × `Cols` méretű táblát készít el a `Bd` listában, és felteszi róla, hogy `Mines` db 1-es van benne. Ezek után a `play_mine/2` eljáráson keresztül kijelzi, hogy melyik mező tartalmára kíváncsi, erre a felhasználónak az `m.` karaktersorozattal kell válaszolnia, ha akna van ott, üres mező esetén pedig meg kell adni, hogy hány akna található a mező körül. Ha a felhasználó válasza `m.`, akkor a program szomorúan konstatálja, hogy veszített (`Mine!`), ha viszont egy szám, akkor a tippelt mező körül lévő mezőkre a `process_ans/5` eljárás második klózával felveszi a megfelelő számossági korlátot. A következő tipp kiválasztását a `select_field/5` eljárás végzi, ez mindig egy olyan mezőt választ ki, amelyről a program egyértelműen tudja, hogy üres (1. klóz), ha ilyet nem talál, akkor pedig egy olyat, amelyről nem tudja biztosan, hogy aknát tartalmaz (2. klóz). Egy egyszerű példajáték (3 × 3-as tábla, akna a 2. sor 1. mezőjén és a 3. sor 3. mezőjén van):

```
| ?- mine(3,3,2,Bd).
% az első két lépésben a program csak reménykedik abban, hogy
% nem lép aknára
Row 1, col 1 (m for mine)? 1.
Row 1, col 2 (m for mine)? 1.
% itt a program már rájön, hogy az (1,3) és a (2,3) mező üres
Row 1, col 3 (m for mine)? 0.
% ebből már azt is tudja, hogy a (2,2) mező üres, így viszont
% a (2,1) mezőn akna van
```

```
Row 2, col 2 (m for mine)? 2.
% mivel a (2,3) mezőről már előzőleg kikövetkeztette, hogy üres,
% ezért azt is meg meri kérdezni
Row 2, col 3 (m for mine)? 1.
% mivel a (2,2) mező körül már csak 1 akna helyét nem tudjuk,
% de azt is tudjuk, hogy a (2,3) mező körül is 1 akna van, ezért
% a (3,1) mezőn nem lehet akna
Row 3, col 1 (m for mine)? 1.
% így viszont a (3,2) mező is üres
Row 3, col 2 (m for mine)? 2.
Bd = [[0,0,0],[1,0,0],[0,0,1]] ? ;
no
```

Vegyük észre, hogy a program inkonzisztens adatok esetén meghiúsulást produkál:

```
| ?- mine(3,3,2,Bd).
Row 1, col 1 (m for mine)? 1.
Row 1, col 2 (m for mine)? 1.
Row 1, col 3 (m for mine)? 0.
Row 2, col 2 (m for mine)? 1.
no
```

A `clpb` mohósága miatt azonban a program egy 20×20-as aknamezőre már nagyon hosszú ideig veszi fel a `mine/4`-ben szereplő `sat(card...)` korlátot (mivel a `card`-ot is „polinom” formára vezeti vissza a listában szereplő összes változó esetén), ilyenkor a program szinte játszhatatlanul lassú lesz.

5. fejezet

A SICStus `clpfd` könyvtára

A következő fejezetben a SICStus `clpfd` könyvtárával fogunk foglalkozni. A `clpfd` könyvtárat az alábbi módon lehet használatba venni:

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

5.1. A `clpfd` könyvtár általános jellemzése

A `clpfd` könyvtár az egész számok véges tartományain (*finite domain, FD*) alapuló CLP rendszert valósít meg. A könyvtár általános alapelve, hogy a CLP relációkat a `#` jellel kell kezdeni, ezzel megkülönböztetve meg őket a hagyományos Prolog jelektől.

Felhasználható függvények

- kétargumentumúak: `+, -, *, /, mod, min, max`
- egyargumentumú: `abs`
  Ezek a hagyományos matematikai műveletekkel megegyező funkcióval rendelkeznek.

Felhasználható relációk

- aritmetikaiak: `#<, #>, #=<, #>=, #= #\=`
  Ezek a jól ismert Prolog relációjelek megfelelői, mindegyik `xfx 700` típusú operátor.
- halmazműveletek:
  - `X in Halmaz` — *X* értékét *Halmaz*-ból veszi
  - `domain([Változók,…], Min, Max)` — *Változók* minden változója értékét a *Min..Max* intervallumból veszi

  ahol *Halmaz* lehet:

  - felsorolás: `{Szám,…}`
  - intervallum: `Min..Max` (`xfx 550` operátor)
  - két halmaz metszete: `Halmaz /\ Halmaz` (`yfx 500` beépített operátor)

– két halmaz uniója: *Halmaz* \/ *Halmaz* (`yfx 500` beépített operátor)

– egy másik halmaz komplemense: *Halmaz* (`fy 500` operátor)

Min-re megengedett az `inf` névkonstans, ami az alsó korlát hiányát jelenti ($-\infty$), hasonlóan *Max*-ra megengedett a `sup` névkonstans, ami pedig a felső korlát hiányát jelenti ($+\infty$). A végtelen korlátok általában csak kényelmi célokat használnak abban az esetben, ha a tényleges korlátok kikövetkeztethetőek. Effektíven végtelen korlátokkal rendelkező változóknak nem sok értelmük van, mert azok a címkézés (ld. később) során végtelen választási pontot hoznának létre (éppen ezért nem lehet olyan változókat címkézni, amelyek végtelen tartománnyal rendelkeznek).

A `clpfd` világban egyszerű korlátoknak csak az *X* $\in$ *Halmaz* jellegű korlátokat tekintjük, minden más összetett korlátnak számít, éppen ezért nagyon nagy hangsúly van az összetett korlátok erősítő tevékenységén (ellentétben a `clpb`-vel, ahol nem is voltak összetett korlátok). Ez a tény (illetve a `clpfd` „lustasága”) teszi lehetővé azt, hogy nagyobb problémákat is megoldjunk vele. Az összetett korlátok erősítő tevékenysége a mesterséges intelligencia-kutatások CSP (Constraint Satisfactory Problems) ágának módszerein alapul.

Egy egyszerű `clpfd` 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 ?

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

A második példán lustaságon kaphatjuk rajta a `clpfd` következtető mechanizmust: ugyan kikövetkeztethető lenne, hogy `Z` csak 20 és 40 közötti *páros* szám lehet (mivel `Y` páros, és `X` 10 és 20 között van), sőt még az is, hogy `Z` semmiképp nem lehet 30 (mert `X` sem lehet 15), de ezt a `clpfd` nem teszi meg, helyette egyszerűen megnézi `X` alsó és felső határát (10 és 20), ezeket beszorozza `Y` alsó és felső határával (ez jelen esetben azonos: 2 és 2), majd az így kapott négy szám minimuma és maximuma által megadott intervallumra szűkíti `Z`-t. Ezt a mechanizmust *intervallum-szűkítés*nek nevezzük, és az 5.6 fejezetben részletesen foglalkozunk majd vele.

5.2. A `clpfd` feladatok megoldási struktúrája

Minden `clpfd` feladat megoldása hasonló struktúrájú programot eredményez, ezért érdemes megkülönböztetnünk a megoldási folyamat fő lépéseit:

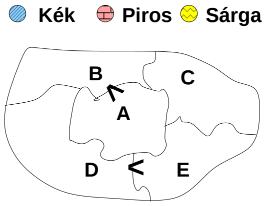
1. **A probléma leképezése a `clpfd` világra**
   Ebben a lépésben a problémának egy olyan modelljét kell megalkotnunk, amelyben a probléma egyes elemeit `clpfd` fogalmakra (változókra, értéktartományokra) képezzük le.

2. **Változók és korlátok felvétele**
   Ebben a lépésben be kell vezetnünk a feladatban szereplő változókat, és fel kell vennünk a változók között fennálló korlát-relációkat.

3. **Címkézés**
   Ha a problémának a korlátok alapján nincs egyértelmű megoldása, vagy ezt a rendszer nem tudta kikövetkeztetni, akkor a változókat el kell kezdenünk szisztematikusan az értéktartományaik

egy-egy lehetséges értékéhez kötni, így meg fogjuk kapni a probléma összes megoldását. A címkézési folyamat a `clpb` könyvtárnál látotthoz hasonló módon működik, de itt egy változó nem csak kétfajta értéket vehet fel. Ha a problémának a korlátok felvétele után már egyértelmű a megoldása, akkor a címkézési fázis elmarad.

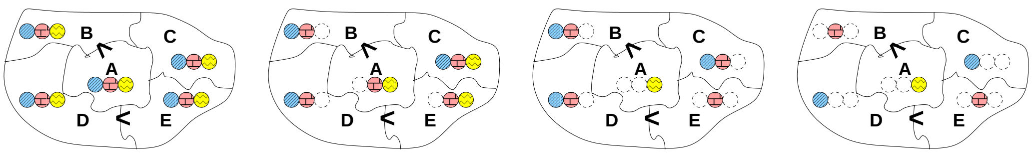
Lássuk a fent elmondottakat egy konkrét példán! A feladat az alábbi térkép kiszínezése kék, piros és sárga színekkel úgy, hogy a szomszédos országok különböző színűek legyenek, és ha két ország határán a `<` jel van, akkor a két szín ábécé-rendben a megadott módon kövesse egymást.



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

1. Minden mezőben elhelyezzük a három lehetséges színt (*változók és tartományaik felvétele*).
2. Az „A” mező nem lehet kék, mert annál „B” nem lehetne kisebb. A „B” nem lehet sárga, mert annál „A” nem lehetne nagyobb. Az „E” és „D” mezők hasonlóan szűkíthetők (*szűkítés, él-konzisztencia biztosítása*).
3. Ha az „A” mező piros lenne, akkor mind „B”, mind „D” kék lenne, ami ellentmondás (*globális korlát, ill. borotválási technika*). Tehát „A” sárga. Emiatt a vele szomszédos „C” és „E” nem lehet sárga (*él-konszitens szűkítés*).
4. „C” és „D” nem lehet piros, tehát kék, így „B” csak piros lehet (*él-konszitens szűkítés*). Tehát az egyetlen megoldás:
   A = sárga, B = piros, C = kék, D = kék, E = piros.

Az alábbi ábrasorozaton láthatóak az egyes lépésekhez tartozó állapotok:



1. lépés 2. lépés 3. lépés 4. lépés

5.3. A CSP problémakör áttekintése

Mint említettük, a `clpfd` könyvtár a mesterséges intelligencia CSP megoldási módszerein alapul, ezért mielőtt továbbmennénk, érdemes áttekinteni a CSP problémakör fogalmait és eredményeit.

A CSP fogalma

- Egy CSP-t egy (X, D, C) hármassal jellemezhetünk, ahol
  - $X = \langle x_1, \ldots, x_n \rangle$ — változók

- $D = \langle D_1, \ldots, D_n \rangle$ — tartományok, azaz nem üres halmazok
- x_i változó a D_i véges halmazból (x_i tartománya) vehet fel értéket ($\forall i$-re $x_i \in D_i$)
- C a problémában szereplő korlátok (atomi relációk) halmaza, argumentumaik X változói (például $C \ni c = r(x_1, x_3)$, $r \subseteq D_1 \times D_3$)

• A CSP feladat megoldása: minden x_i változóhoz egy $v_i \in D_i$ értéket kell rendelni úgy, hogy minden $c \in C$ korlátot egyidejűleg kielégítsünk.

5.3.1. definíció: egy c korlát egy x_i változójának d_i értéke *felesleges*, ha nincs a c többi változójának olyan értékrendszere, amely $x_i = d_i$-vel együtt kielégíti c-t.

5.3.1. tétel: felesleges érték elhagyásával (szűkítés) ekvivalens CSP-t kapunk.

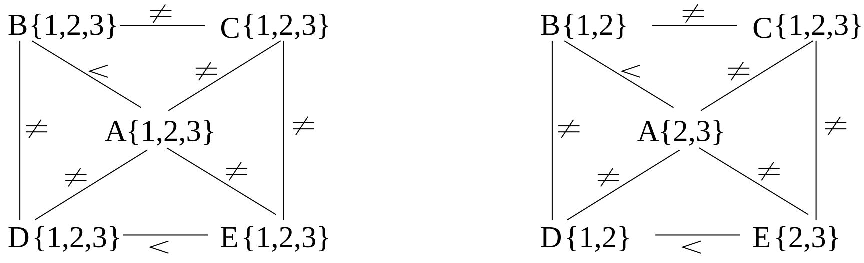
5.3.2. definíció: egy korlát *élkonzisztens* (*arc consistent*), ha egyik változójának tartományában sincs felesleges érték. A CSP *élkonzisztens*, ha minden korlátja élkonzisztens. Az élkonzisztencia szűkítéssel biztosítható.

Az *élkonzisztencia* elnevezés onnan ered, hogy ha minden reláció bináris, akkor a CSP probléma egy gráffal ábrázolható, ahol minden változónak egy csomópont, minden relációnak egy él felel meg.

A CSP megoldás folyamata

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

A térképszínezés, mint CSP feladat esetén minden országhoz egy változót rendeltünk hozzá, ennek a változónak az értéke fogja az ország színét kódolni. A színekhez ábécésorrend szerint az 1, 2, 3 értékek valamelyikét rendeltük hozzá (kék $\to$ 1, piros $\to$ 2, sárga $\to$ 3), majd felvettük a korlátokat egyrészt arra, hogy a szomszédos országok színei különböznek (ez a változóértékek világában egy $\neq$ típusú relációt jelent), másrészt arra, hogy az országok színei között megadott $<$ relációk is teljesüljenek. Ezzel kaptunk egy kiinduló korlát-gráfot, amit a felesleges élek elhagyásával szűkítettünk. Az alábbi ábrán látható a kiinduló korlát-gráf és annak élkonzisztens szűkített változata:



A CSP sémának a CLP világba történő beágyazásával kapjuk a SICStusban lévő `clpfd` könyvtárat. Minden CSP változónak egy `clpfd` változó feleltethető meg, a CSP változók értéktartományainak pedig egy-egy `clpfd` egyszerű korlát. A többi CSP korlát összetett `clpfd` korlátként jelenik meg. A `clpfd` korlát-tár új változótartomány felvételén vagy egy meglévő változó tartományának szűkítésén módosulhat. Az összetett korlátok *démon*ok lesznek, amelyek hatásukat az *erősítés*en keresztül fejtik ki (ld. 1.3 fejezet). Az erősítés mindig egyszerű korlátokat ad a korlát-tárhoz. A démonok ciklikusan működnek: megszületnek, szűkítenek, elalszanak, aktiválódnak, szűkítenek, elalszanak, …aktiválódnak, szűkítenek, és amikor már levezethetőek a korlát-tár tartalmából, akkor megszűnnek létezni. A démonokat mindig az érintett korlátbeli változók tartományának módosulása aktiválja. A szűkítés mértéke a démontól függ, néha nem előnyös az összes lehetséges szűkítést elvégezni, mert túlságosan költséges lenne.

5.4. A `clpfd` könyvtár jellegzetességei

Ebben az alfejezetben a `clpfd` könyvtár néhány jellegzetességét mutatjuk be példákon keresztül. A példák megértéséhez egyetlen, nagyon fontos állítást kell szem előtt tartanunk: **a `clpfd` démonok csak a korlát-táron keresztül hatnak egymásra!**

A fenti mondat azt takarja, hogy a démonok nem „látják” egymást, az egymással való interakciójukat kizárólag a korlát-táron keresztül végzik: az egyik démon szűkíti a korlát-tárat, ennek hatására egy másik démon felébred, szűkít, erre egy harmadik démon ébred fel és így tovább... Előfordulhatnak azonban olyan esetek, amikor egyik démon sem tud felébredni, és így esetleg egy nyilvánvaló ellentmondást nem vesz észre a rendszer, mint például a következő példában:

```
| ?- domain([X,Y,Z], 1, 2), X #\= Y, X #\= Z, Y #\= Z.
X in 1..2,
Y in 1..2,
Z in 1..2 ? ;
no
```

Mivel `X`, `Y` és `Z` értékkészlete is az 1 és a 2 számokból áll, és az `X #\= Y` jellegű korlátok démonai csak akkor ébrednek fel, ha valamelyik változójuk behelyettesített lesz, ezért egyik démon sem tud szűkíteni, és az ellentmondás nem derül ki. A megoldást globális korlátok (pl. az `all_distinct/1`) használata jelenti majd. A globális korlátok olyan korlátok, amelyek működésükkel több korlát hatását fogják össze egyetlen démonban, így ez a démon rá tud jönni az ilyen jellegű ellentmondásokra. Ezekről a korlátokról a későbbiekben még részletesen lesz szó (ld. 5.19.1. fejezet).

Hasonló szituációt jelent a következő példa is:

```
| ?- X #> Y, Y #> X.
Y in inf..sup,
X in inf..sup ? ;
no
```

Ha ugyanezt a két korlátot úgy vesszük fel, hogy közben `X` és `Y` tartományát végesre szűkítjük, akkor már nem jelentkezik a probléma:

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

Azonban ha a tartományt egy picit tágabbra vesszük, újabb problémával találjuk szembe magunkat, a meglepően nagy futási idővel:

```
| ?- statistics(runtime,_),
     ( domain([X,Y], 1, 100000), X #> Y, Y #> X
     ; statistics(runtime,[_,T])
     ).
T = 3630 ? ;
no
```

Ennek oka ismét abban keresendő, hogy a démonok csak a korlát-táron keresztül hatnak egymásra. Nézzük meg ugyanis ennek a példának a futását az `fdbg` nyomkövető könyvtár használatával, 10-es tartományhatárra (az `fdbg` könyvtárról bővebben a 6. fejezetben lesz szó)!

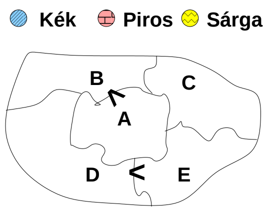
```
| ?- use_module(library(fdbg)).
| ?- fdbg_on, fdbg_assign_name(X, x), fdbg_assign_name(Y, y),
     domain([X,Y], 1, 10), X #> Y, Y #> X.

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

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

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

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

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

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

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

A kimenetet értelmezve láthatjuk, hogy az `X` és az `Y` változók tartománya nagyon lassan szűkül: kezdetben az `X in 1..10` feltétel és az `X #> Y` (a `clpfd` belső ábrázolása szerint `X #>= Y+1`) korlát démona miatt `X` tartománya a `2..10` halmazra, `Y`-é pedig az `1..9`-re szűkül. Ekkor a tartománymódosulások hatására felébred az `Y #> X` korlát démona is, és szűkíti `X`-et a `2..8`, `Y`-t a `3..9` halmazra. Ettől viszont újból felébred az `X #> Y` korlát démona, és így folytatódik a dolog egészen addig, amíg végül az egyik démon észre nem veszi, hogy itt meghiúsulás fog következni. Nagyobb tartományhatárok esetén ez a láncreakció értelemszerűen tovább tart, sőt, mint láttuk, végtelen tartományhatárok esetén nem is indul el.

5.5. Egyszerű constraint feladatok megoldása

5.5.1. Térképszínezés

Emlékeztetőül a feladat: színezzük ki az alábbi térképet kék, piros és sárga színekkel úgy, hogy a szomszédos országok különböző színűek legyenek, és ha két ország határán a < jel van, akkor a két szín ábécé-rendben a megadott módon kövesse egymást.



Először írjuk fel a megfelelő korlátokat leíró `clpfd` célsorozatot:

```
| ?- 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
```

Látható, hogy a Prolog az élkonzisztencia biztosítását elvégezte, de a megoldást még nem tudta kikövetkeztetni. A megoldások meghatározásához meg kell kérni a rendszert, hogy az `A` változót rendre helyettesítse be az 1, 2, 3 értékekre. Ez többféleképpen is elvégezhető: egyrészt a hagyományos `member/2` eljárással, amelynél azonban egyesével fel kell sorolnunk `A` lehetséges értékeit, ami nagy értékkészletnél kényelmetlen lehet:

```
| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E,
     member(A, [1,2,3]).
A = 3, B = 2, C = 1, D = 1, E = 2 ?
```

Az ilyen problémák megoldására szolgál az `indomain/1` eljárás, amely ezt a behelyettesítést automatikusan elvégzi a paraméterként adott változóra:

```
| ?- domain([A,B,C,D,E], 1, 3), …, indomain(A).
A = 3, B = 2, C = 1, D = 1, E = 2 ?
```

Előfordulhatott volna azonban, hogy `A` behelyettesítése még mindig nem elég ahhoz, hogy kiderüljön az összes megoldás, ezért célszerű a behelyettesítést mind az 5 változóra elvégezni. Ezt a `labeling/2` eljárás végzi, amely első paramétere egy opciólista, amely a címkézés menetét szabályozza (ezzel majd később foglalkozunk), második paramétere pedig egy változólista, amelyre a címkézést el akarjuk végezni.

```
| ?- domain([A,B,C,D,E], 1, 3), …, labeling([],[A,B,C,D,E]).
A = 3, B = 2, C = 1, D = 1, E = 2 ?
```

Annak megfogalmazására, hogy az `A`, `C` és `E` változók értéke mind különbözik, a Prolog kínál egy egyszerűbb és hatékonyabb megoldást is, az `all_distinct/1` predikátumot. Ez egy listát vár paraméterként, és ügyel arra, hogy a lista összes eleme különböző értékeket vegyen fel.

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

Látható, hogy itt már címkézésre se volt szükség, mivel az `all_distinct` korlát „erősebb” a páronkénti különbözőségnél.

5.5.2. Kódaritmetika (SEND+MORE=MONEY)

A feladvány: írjon a betűk helyébe tízes számrendszerbeli számjegyeket (azonosak helyébe azonosakat, különbözőek helyébe különbözőeket) úgy, hogy a SEND+MORE=MONEY egyenlőség igaz legyen. Szám elején nem lehet 0 számjegy.

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

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

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

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

A fenti programon jól látszik a szokásos `clpfd` struktúra (tartományok felvétele, korlátok felvétele, címkézés). Az „azonos betűk azonos számokat, különböző betűk különböző számokat jelentenek” kitételt egy saját `alldiff/1` predikátummal valósítjuk meg, amely lényegében megegyezik az `all_different/1` beépített globális korláttal. Fontos megjegyezni, hogy az `all_different/1` és az `all_distinct/1` korlátok *nem* ekvivalensek, az előbbi csak páronkénti különbözőséget valósít meg! Gyakran azonban ez is elég, és az `all_different` emellett gyorsabb futást biztosít, tehát néha érdemes használni [1]. Ezen magyarázat után lássuk a program futását:

```
| ?- send(SEND, MORE, MONEY).
MORE = 1085, SEND = 9567, MONEY = 10652 ? ;
no
```

Nézzük meg azt is, hogy mit adna ki a program, ha elhagynánk a címkézést:

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

Amint látható, a rendszer helyből rájött arra, hogy mivel az eredmény öt számjegyből áll, a két összeadandó viszont csak négyből, ezért `M` csak 1 lehet. Ha viszont `M` 1, akkor `S`-nek 9-nek kell lennie, különben nem keletkezhet átvitel, ilyenkor viszont `O` csak nulla lehet. A többi változó értéke csak a címkézés után derül ki.

5.5.3. A zebra feladat

Szintén egy `clpfd`-ben könnyen megfogható feladat: adott 5 különböző nemzetiségű ember, 5 különböző színű ház, 5 különböző foglalkozás, 5 különböző állat és 5 különböző ital. Egy embernek pontosan egy foglalkozása, nemzetisége, háza, kedvenc itala és állata van. A feladat az, hogy az ismert kötöttségek alapján megmondjuk, hogy melyik nemzetiségű ember kedvenc állata a zebra. Az embereket megsorszámozzuk 1-től 5-ig, majd minden egyes foglalkozáshoz, nemzetiséghez, házhoz, italhoz és állathoz egy constraint változót rendelünk, amely értékét az 1..5 halmazból veszi fel. Egy ilyen változó i értéke azt jelenti, hogy az általa reprezentált tulajdonság az i. számú emberhez tartozik. A kötöttségeket ez alapján könnyen felírhatjuk, például az, hogy a diplomata háza a sárga, egyszerűen a két megfelelő constraint-változó egyenlővé tételével biztosítható. A kötöttségek a programból könnyen kideríthetők, itt nem soroljuk fel őket. Az egyetlen figyelmet érdemlő feltétel annak a megvalósítása, hogy valaki egy adott ház szomszédjában lakik. Ezt a `nextto/2` predikátum kezeli. A `nextto/2` tulajdonképpen egy vagylagos szerkezet, amit a Prolog választási pontokkal valósít meg - vagyis spekulatív módon veszi fel a vagy-kapcsolatban szereplő korlátokat, és minden lehetőségre megpróbálja megoldani az aktuális korlát rendszert. Ez nem a legszerencsésebb megoldás, mivel sok választási pont esetén rengeteg időt igényelhet. A vagylagos szerkezetekről a `clpfd` esettanulmányokban (108. oldal) még lesz szó. Adott esetben a

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

megoldás sokkal szerencsésebb lenne, mivel ez nem generál választási pontokat.

```
:- 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, [england,spain,japan,norway,italy], ZOwner).

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

5.5.4. N királynő a sakktáblán

A feladat elhelyezni egy $n * n$-es sakktáblán n királynőt úgy, hogy egyik se üsse a másikat.

A megoldás minden királynőt külön oszlopba tesz, majd mindegyikükhöz egy változót rendel, ami a királynő oszlopbeli pozícióját adja meg. A változók tartományának deklarálása után minden két királynő között felállít egy constraint-et (`no_threat/3`), ami azt adja meg, hogy a két királynő nem üti egymást. Ehhez meg kell adni a két királynő oszlopainak távolságát, ami itt az `I` paraméter. Ezek után a program végrehajtja a címkézést first-fail heurisztikával. A first-fail elv mindig a legkisebb értékkészlettel rendelkező változót helyettesíti be először, abban reménykedve, hogy így hamarabb kiderülnek a hibás ágak. A címkézési módokról később lesz szó.

```
:- 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.
```

5.6. Szűkítési szintek

A könnyebb megértés érdekében először informálisan, egy egyszerű kétargumentumú relációra fogjuk megfogalmazni az *intervallum-szűkítés* és a *tartományszűkítés* fogalmát, utána pedig formalizáljuk a leírtakat.

Tekintsünk egy $r(X, Y)$ bináris relációt! Ekkor r *tartományszűkítése* során X tartományából elhagyjuk az összes olyan x értéket, amelyhez nem található Y tartományában olyan y érték, hogy $r(x, y)$ fennáll. Hasonlóan szűkítjük Y tartományát is. A folyamat eredménye az élkonzisztencia (lásd a fogalom definícióját a 5.3 fejezetben). *Intervallum-szűkítés* során viszont X tartományából elhagyjuk annak alsó vagy felső határát, ha ahhoz nem található olyan y érték, amely Y határai közé esik, és azzal az r reláció fennáll. Hasonlóan szűkítjük Y tartományának határait is, és ezeket a lépéseket addig ismételjük, amíg tudunk szűkíteni. Ez a módszer nem biztosít élkonzisztenciát, de gyorsabban elvégezhető.

Példa: legyen $x \in \{0, 1, 2, 3, 4, 5\}$ és $y \in \{-1, 1, 3, 4\}$, $r(x, y)$ pedig az $x = |y|$ reláció. A tartományszűkítés x értékkészletéből elhagyja a 0, 2, 5 értékeket, hiszen semelyik y érték abszolútértéke nem lehet sem 0, sem 2, sem 5. Az intervallum-szűkítés viszont először csak a 0 és az 5 kizárásával próbálkozik. 0-t nem zárhatja ki, hiszen y tartományának szélső értékei közé (tehát -1 és 4 közé) esik a 0, és $0 = |0|$. 5-öt kizárhatja, mert sem a -5, sem az 5 nem esik y szélső értékei közé, de a 4-et már nem zárhatja ki, ezért x csak a $\{0, 1, 2, 3, 4\}$ halmazra szűkül.

A fenti példán jól látszik az intervallum-szűkítés két gyengesége:

1. csak a tartomány szélső értékeit hajlandó elhagyni, ezért nem hagyja el a `2` értéket;
2. a másik változó tartományában nem veszi figyelembe a „lyukakat”, így a példában `Y` tartománya helyett annak *lefedő intervallumát*, azaz a `-1..4` intervallumot tekinti, ezért nem hagyja el `X`-ből a `0` értéket.

Ugyanakkor az intervallum-szűkítés általában konstans idejű művelet, míg a tartományszűkítés ideje (és az eredmény mérete) erősen függ a tartományok méretétől, ezért sok esetben a SICStus `clpfd` könyvtár csak az intervallum-szűkítést garantálja, a tartományszűkítést nem.

Ezek után fogalmazzuk meg a definícióinkat formálisan is!

Jelölések

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

A szűkítési szintek definíciója

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

A tartományszűkítés lokálisan (egy korlátra nézve) a lehető legjobb, de nem garantálja a megoldást akkor sem, ha az összes korlát tartományszűkítő, mivel nem tudja figyelembe venni a többi korlát hatását. Ezt illusztrálja a már bemutatott `all_different` $\longleftrightarrow$ `all_distinct` probléma, ahol kihasználjuk, hogy az `all_different` ekvivalens a páronkénti különbözőségek felvételével:

```
| ?- domain([X,Y,Z], 1, 2), X #\= Y, Y #\= Z, Z #\= X.
X in 1..2, Y in 1..2, Z in 1..2 ? ;
no
| ?- domain([X,Y,Z], 1, 2), all_distinct([X,Y,Z]).
no
```

A SICStusban a halmazkorlátok (triviálisan) tartományszűkítők. A *lineáris* aritmetikai korlátok legalább intervallum-szűkítők, a nemlineáris aritmetikai korlátokra nincs garantált szűkítési szint. Ha a változók valamelyik határa végtelen (`inf` vagy `sup`), akkor nincs garantált szűkítési szint, de az aritmetikai és a halmazkorlátok ilyenkor is szűkítenek. A később tárgyalt korlátokra egyenként megadjuk majd a szűkítési szinteket.

Néhány példa a szűkítési szintekre:

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

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

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

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

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

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

5.7. Korlátok végrehajtása

Egy korlát végrehajtása több fázisból áll:

1. A korlát kifejtése belső, elemi korlátokra (ld. 5.14. fejezetben)
2. A korlát felvétele. Itt rögtön két lehetőség adódik:
   - Egyszerű korlát (pl. `X #< 4`) esetén a korlát azonnal végrehajtásra kerül
   - Összetett korlát esetén a korlátból démon képződik, a démon elvégzi a lehetséges szűkítéseit, meghatározza, hogy milyen feltételek esetén kell újra aktiválódnia, majd elalszik.
3. Ha a korlátból démon képződött, és a démon ébresztési feltételei teljesülnek, akkor aktiválódik, elvégzi a szűkítéseit, majd dönt a folytatásról. A döntés eredménye kétféle lehet:
   - Ha a démon már levezethető a tárból, akkor befejezi működését
   - Ha a démon még nem vezethető le a tárból, akkor újból elalszik

Nézzük az eddig elmondottakat néhány konkrét példán!

`A #\= B` (tartományszűkítő)

- **Aktiválás feltétele:** ha `A` vagy `B` konkrét értéket kap
- **A szűkítés módja:** az adott értéket kizárja a másik változó értelmezési tartományából
- **A folytatás menete:** mivel ilyenkor már a démon biztosan levezethető a tárból, ezért a démon működése befejeződik

`A #< B` (tartományszűkítő)

- **Aktiválás feltétele:** ha `A` alsó határa (min(`A`)) vagy `B` felső határa (max(`B`)) változik
- **A szűkítés módja:** `A` tartományából kihagyja az $X \geq \max(\texttt{B})$ értékeket, `B` tartományából pedig kihagyja az $Y \leq \min(\texttt{A})$ értékeket

- **A folytatás menete:** ha max(`A`) < min(`B`), akkor lefut, egyébként elalszik

`all_distinct([A`$_1$`,A`$_2$`,…])` (tartományszűkítő)

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

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

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

- **A folytatás menete**: ha már csak egy nem-konstans argumentuma van, akkor lefut, különben újra elalszik. Látszólag jobb döntésnek tűnhet, ha a korlát akkor futna le, amikor a tartományok már mind diszjunktak, de a SICStus nem így csinálja, valószínűleg azért, mert nem éri meg.

`X+Y #= T` (intervallum-szűkítő)

- **Aktiválás feltétele**: ha bármelyik változó alsó vagy felső határa változik (az intervallum-szűkítés miatt nem minden tartományváltozásra ébred fel)
- **A szűkítés módja**: `T`-t szűkíti a `(`min `X+`min `Y)..(`max `X+`max `Y)` intervallumra, `X`-t szűkiti a `(`min `T-`max `Y)..(`max `T-`min `Y)` intervallumra, `Y`-t analóg módon szűkíti.
- **A folytatás menete**: ha (a szűkítés után) mindhárom változó konstans, akkor lefut, különben újra elalszik.

Mivel a `clpfd` alapvetően „lusta” működésű, és nem végez el minden lehetséges szűkítést, ezért előfordulhat, hogy ugyanazon változókra megfogalmazott, jelentéstartalomban megegyező, de különböző szintaktikájú korlátok nem azonos mértékben szűkítenek, mint ahogy azt a következő példa is mutatja:

```
| ?- domain([X,Y], 0, 100), X+Y #=10, X-Y #=2.
                X in 2..10, Y in 0..8 ?

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

Az alsó és a felső példának ugyanaz a megoldása, az első esetben az `X-Y #= 2` korlát intervallum-szűkítő, és az intervallum-szűkítés a már fennálló `X in 2..10` és `Y in 0..8` tartományokat nem tudja tovább szűkíteni. Az `X+2*Y #= 14` korlát viszont ugyan garantáltan nem tartományszűkítő, de itt mégis elvégzi a tartományszűkítést, és ezzel megtalálja a megoldást.

5.8. Korlátok tükrözése: *reifikáció*

5.8.1. definíció: egy C korlát *reifikáció*ja (*tükrözés*e) a korlát igazságértékének megjelenítése egy 0-1 értékű korlát változóban. Jelölése: C `#<=> B`. Ezt úgy kell értelmezni, hogy `B` egy 0-1 értékű változó, és `B` akkor és csak akkor 1, ha C igaz.

Példa: `(X #>= 3) #<=> B` jelentése: `B` az `X` $\geq$ `3` egyenlőtlenség igazságértéke.

Az eddig ismertetett halmaz- és aritmetikai korlátok (az úgynevezett *formulakorlátok*) mind tükrözhetőek, de a globális korlátok (pl. `all_different/1`, `all_distinct/1`) nem. A 5.19.2. fejezetben ismertetésre kerülő FD predikátumok a felhasználó határozza meg az FD predikátum klózainak megfelelő kialakításával.

Egy C `#<=> B` korlát végrehajtása többféle szűkítést is igényel:

a) amikor `B`-ről kiderül valami (azaz behelyettesítődik): ha `B=1`, fel kell venni (*post*) a korlátot, ha `B=0`, fel kell venni a negáltját.

b) amikor C-ről kiderül, hogy levezethető a tárból, végre kell hajtani a `B=1` helyettesítést

c) amikor $\neg C$-ről kiderül, hogy levezethető a tárból, végre kell hajtani a `B=0` helyettesítést

A fenti három fajta szűkítést három különböző démon végzi. A levezethetőségi vizsgálat különböző „ambíciókkal”, különböző bonyolultsági szinteken végezhető el (bővebben: 5.9. fejezet).

Lássuk a fent leírtakat működés közben!

- Alappélda, csak `B` szűkül:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.9. Levezethetőségi szintek

A SICStus Prolog kétfajta levezethetőségi szintet ismer, a tartományszűkítés és az intervallum-szűkítés fogalmához hasonlóan:

5.9.1. definíció: a C n-változós korlát *tartomány-levezethető* az s tárból, ha változóinak s-ben megengedett tetszőleges $V_j \in D(X_j, s)$ értékkombinációjára $(j = 1, \dots, n)$ $C(V_1, \dots V_n)$ fennáll.

5.9.2. definíció: a C n-változós korlát *intervallum-levezethető* az s tárból, ha változóinak s-ben megengedett tetszőleges $V_j \in D'(X_j, s)$ értékkombinációjára $(j = 1, \dots, n)$ $C(V_1, \dots V_n)$ fennáll.

A fentiekből a $D(x_j, s) \subseteq D'(x_j, s)$ relációt figyelembe véve következik, hogy ha C intervallum-levezethető, akkor tartomány-levezethető is. A kétféle levezethetőségi vizsgálatra azért van szükség, mert a tartomány-levezethetőség vizsgálata általában bonyolultabb (és tovább is tart), mint az intervallum-levezethetőségé. Például az `X #\= Y` korlát tartomány-levezethető, ha `X` és `Y` tartományai diszjunktak (a tartományok méretével arányos költség), ugyanakkor az intervallum-levezethetőséghez elég az `X` és `Y` tartományainak lefedő intervallumait vizsgálni (ami konstans költségű művelet).

A SICStus-ban a tükrözött halmazkorlátok kiderítik a tartomány-levezethetőséget, a tükrözött *lineáris* aritmetikai korlátok legalább az intervallum-levezethetőséget (egyes esetekben a tartomány-levezethetőséget is, lásd az előző alfejezet utolsó 4 példáját). A tükrözött nemlineáris aritmetikai korlátokra még az intervallum-levezethetőség kiderítése sem garantálható.

5.10. Egy bonyolultabb `clpfd` példa: mágikus sorozatok

5.10.1. definíció: egy $L = (x_0, x_1, \dots, x_{n-1})$ sorozat *mágikus* ($x_i \in [0..n-1]$), ha minden $i \in [0..n-1]$-re L-ben az i szám pontosan x_i-szer fordul elő.

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

A feladat `clpfd` megoldásához definiálni fogunk a sorozatok között egy transzformációt:

5.10.2. definíció: az $X = (x_0, x_1, \dots, x_{n-1})$ sorozatnak az $Y = \mathcal{E}(X) = (y_0, y_1, \dots, y_{n-1})$ sorozat az *előfordulás-sorozat*a, ha minden $i \in [0..n-1]$-re X-ben az i szám pontosan y_i-szer fordul elő.

5.10.1. Egyszerű `clpfd` megoldás

Látható, hogy a mágikus sorozatok azok a sorozatok, amelyek erre a $\mathcal{E}$ transzformációra nézve fixpontok (azaz olyan sorozatok, amelyeket a transzformáció önmagába visz át). Ennek felhasználásával a feladatra könnyen adható egy néhány soros `clpfd` megoldás:

```
% 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, Ei+1, …], i, Sor): Sor-ban az i
% szám Ei-szer, az i+1 szám Ei+1-szer stb. fordul elő.
% Ez a predikátum valósítja meg a fenti előfordulás-sorozat transzformációt
elofordulasok([], _, _).
elofordulasok([E|Ek], I, Sor) :-
        pontosan(I, Sor, E),
        J is I+1, elofordulasok(Ek, J, Sor).

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

% outof(I, L): Az I szám L-ben nem fordul elő.
outof(_, []).
outof(X, [Y|Ys]) :- X #\= Y, outof(X, Ys).
```

Példafutás (csak a `pontosan/3` hívások érdekelnek minket):

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

```
?+      1      1 Exit: pontosan(0,[3,0,0,0],3) ? z
(...)
?+      1      1 Exit: pontosan(0,[2,0,_D,0],2) ?
```

5.10.2. Redundáns korlátok bevezetése

A fenti változat hatékonyságán sokat segíthetünk *redundáns korlátok* felvételével. A redundáns korlátok olyan korlátok, amelyek formálisan nem közölnek új információt a feladatról, hanem a már fennálló korlátok következményei, azonban úgy vannak „megfogalmazva”, hogy ezzel segítik a program végrehajtását, mert olyan esetekben is szűkítéseket eredményeznek, amikor a redundancia nélküli változat erre nem volt képes. A redundáns korlátokhoz a következő állítást fogjuk felhasználni:

5.10.1. tétel: ha az $L = (x_0, x_1, \ldots, x_{n-1})$ sorozat mágikus, akkor $\sum_{i<n} x_i = n$ és $\sum_{i<n} i \times x_i = n$.

Ezzel a program fő klóza a következőképpen módosul:

```
% Az L lista egy N hosszú mágikus sorozat
magikus2(N, L) :-
    length(L, N), N1 is N-1, domain(L, 0, N1),
    osszege(L, S),                % ∑i∈ [1..N] Li  =  S
    szorzatosszege(L, 0, SP),     % ∑i∈ [1..N] i ∗ Li  =  SP
    call(S #= N), call(SP #= N), % lásd a megjegyzést
    elofordulasok(L, 0, L).       % lásd az előző lapon

% osszege(L, Ossz): Ossz = ∑i Li
osszege([], 0).
osszege([X|L], X+S) :- osszege(L, S).

% szorzatosszege(L, I, Ossz): Ossz = I ∗ L1 + (I+1) ∗ L2 + ...
szorzatosszege([], _, 0).
szorzatosszege([X|L], I, I*X+S) :-
    J is I+1, szorzatosszege(L, J, S).
```

Az `osszege/2` és a `szorzatosszege/3` hívásokat külön nem fejtjük ki, ezek a megfelelő korlátkifejezéseket *építik fel* az `S` és `SP` változókba. A `call/1` alkalmazására azért van szükség, mert a korlátokat a `clpfd` könyvtár fordítási időben átalakítja a saját, belső formátumára, és ez a felhasználó által futásidőben felépített korlátokra nem történik meg. A megoldást a korlátkifejtési fázis késleltetése jelenti a `call/1` segítségével. Egyébként a programnak ez a változata `N=10` esetben kb. ötvenszer gyorsabb az előző verziónál, és a 4 hosszú mágikus sorozatok közül az első megoldást visszalépés nélkül adja ki!

5.10.3. Tükrözéses megoldás

A tükrözés mechanizmusának felhasználásával egy elegáns, ráadásul hatékonyabb megfogalmazást is adhatunk a `pontosan/3` eljárásra, és ezzel az egész feladatra:

```
magikus3(N, L) :-
        length(L, N),
```

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

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

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

A megoldás lényege, hogy a `pontosan3/3` eljárásban az `L` lista minden `X` elemére felveszünk egy `X #= I` korlátot, és ennek igazságértékét egy `B` változóban tükrözzük. Az összes `B` érték összegének pontosan `E`-vel kell megegyeznie. Ezzel a megoldással sikerült kiszűrni a `pontosan/3` eljárásból az eddig meglévő diszjunkciót, és ezzel sokkal hatékonyabb kódot sikerült készítenünk, ami az alábbi összehasonlító táblázatból is látszik (1 perc időkorlát, Pentium III 600 MHz-es processzor):

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

5.11. Logikai korlátok

Annak ellenére, hogy a `clpfd` könyvtár alapvetően véges, egész értékű tartományok kezelésére használatos, lehetőség van logikai korlátok használatára is. Logikai korlátokat háromféle alkotóelemből építhetünk fel:

- változókból, ilyenkor a változók tartománya automatikusan a 0..1 tartományra szűkül (a 0 a logikai hamis, az 1 a logikai igaz értéket fogja jelenteni)
- tükrözhető aritmetikai- vagy halmazkorlátokból
- más logikai korlátokból

Ezen építőelemek összekapcsolására az alábbi operátorok használatosak:

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

Észrevehetjük, hogy a korlátok igazságértékének tükrözésére használt `#<=>` operátor jelen van a fenti táblázatban is. Ennek az az oka, hogy a tükrözési jelölés valójában a logikai korlát-fogalom speciális esete. Fontos azonban megjegyezni, hogy az *összes* logikai korlát a C `#<=> B` alakú *elemi* korlátra vezetődik vissza. Például:

`X #= 4 #\/ Y #> 6` $\Longleftrightarrow$ `X #= 4 #<=> B1, Y #> 6 #<=> B2, B1+B2 #> 0`

Vigyázat! A diszjunktív logikai korlátok gyengén szűkítenek: egy n-tagú diszjunkció csak akkor tud szűkíteni, ha egy kivételével valamennyi tagjának a negáltja levezethetővé válik (a fenti példában akkor, ha `X #= 4` vagy `Y #=< 6` levezethető lesz).

5.12. Példa a logikai korlátokra: lovagok, lókötők és normálisak

Egy szigeten minden bennszülött lovag, lókötő, vagy normális. A lovagok mindig igazat mondanak, a lókötők mindig hazudnak, a normális emberek pedig néha hazudnak, néha igazat mondanak. Adott három bennszülött, A, B és C, akik közül egy lovag, egy lókötő és egy normális (de nem feltétlenül ebben a sorrendben). Az alábbi állításokat teszik:

A: Én normális vagyok. B: A igazat mond. C: Én nem vagyok normális.

Kérdés: melyikőjük lovag, melyikőjük lókötő és melyikőjük normális?

A `clpfd` megoldás során alkalmazzuk az alábbi kódolást: normális → `2`, lovag → `1`, lókötő → `0` (ez azért jó, mert így minden lovag állítása az 1-es igazságértékbe, és minden lókötő állítása a 0-s igazságértékbe tükrözhető, ami megegyezik a hozzájuk rendelt azonosítóval).

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

% Bevezetünk néhány operátort az állítások egyszerűbb leírására.
:- op(700, fy, nem).       :- op(900, yfx, vagy).
:- op(800, yfx, és).       :- op(950, xfy, mondja).

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

% értéke(A, Érték): Az A állítás igazságértéke Érték.
értéke(X = Y, E) :-
    X in 0..2, Y in 0..2, E #<=> (X #= Y).
értéke(X mondja M, E) :-
    X in 0..2, értéke(M, E0),
```

```
    E #<=> (X #= 2 #\/ E0 #= X).
értéke(M1 és M2, E) :-
    értéke(M1, E1), értéke(M2, E2), E #<=> E1 #/\ E2.
értéke(M1 vagy M2, E) :-
    értéke(M1, E1), értéke(M2, E2), E #<=> E1 #\/ E2.
értéke(nem M, E) :-
        értéke(M, E0), E #<=> #\E0.

| ?- all_different([A,B,C]), A mondja A = 2,
     B mondja A = 2, C mondja nem C =2,
     labeling([], [A,B,C]).
A = 0, B = 2, C = 1 ? ;
no
```

5.13. További globális aritmetikai korlátok

Az eddigiek során már megismerkedhettünk két globális korláttal: az `all_different/1` és az `all_distinct/1` korlátokkal. A SICStus azonban ismer még néhány hasznos globális aritmetikai korlátot is, amelyeket ebben a fejezetben fogunk bemutatni. Ezen korlátok közös jellemzője, hogy a többi globális korláthoz hasonlóan nem tükrözhetőek.

- `scalar_product(Coeffs, Xs, RelOp, Value)`
  Igaz, ha a `Coeffs` egészekből álló együttható-lista és az `Xs` korlát-változókból és egészekből álló lista skalárszorzata a `RelOp` relációban áll a `Value` értékkel (`Value` lehet egész szám vagy korlát-változó). `RelOp` egy tetszőleges aritmetikai összehasonlító operátor (pl. `#=`, `#<` stb.). Intervallum-szűkítést biztosít. Érdekességként megjegyezzük, hogy minden lineáris aritmetikai korlát átalakítható egy megfelelő `scalar_product` hívássá.

- `sum(Xs, RelOp, Value)`
  Igaz, ha az `Xs` korlát-változókból és egészekből álló lista összege a `RelOp` relációban áll a `Value` értékkel (`Value` lehet egész szám vagy korlát-változó). Megegyezik a `scalar_product(Csupa1, Xs, RelOp, Value)` hívással, ahol `Csupa1` csupa 1-es számból álló lista, és a hossza megegyezik `Xs` hosszúságával.

- `knapsack(Coeffs, Xs, Value)`
  Jelentése: `Coeffs` és `Xs` skalárszorzata `Value`. Csak nem-negatív számok és véges tartományú változók megengedettek, viszont cserébe tartomány-konzisztenciát biztosít.

A 5.5.2. fejezetben ismertetett kódaritmetika példa egyszerűbb megoldása `scalar_product/4` segítségével:

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

```
        scalar_product([10000|Pow10], [M,O,N,E,Y],
                       #=, MONEY),
    %   MONEY #= 10000*M+1000*O+100*N+10*E+Y,
        SEND+MORE #= MONEY.
```

5.14. A formulakorlátok belső megvalósítása

5.14.1. definíció: *formulakorlát*oknak nevezzük az összes operátoros jelöléssel írt korlátot, tehát az eddig tárgyalt összes korlátot, a globális korlátok kivételével.

Mint azt már a 5.7. fejezet elején említettük, a formulakorlátokat a SICStus `clpfd` könyvtára elemi korlátok konjunkciójára fordítja le a `goal_expansion/3` kampó-eljárás segítségével. Ez az eljárás minden formulakorlátot fordítási időben a `scalar_product/4` korlátra és/vagy nem publikus elemi korlátokra fejt ki. Az átfordítás futásidőig elhalasztható a korlát `call/1`-be való ágyazásával, ez a futás közben felépített formulakorlátok esetén hasznos.

A legfontosabb elemi korlátok a `clpfd` modulban

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

A legtöbb elemi korlát *pontszűkítő*. A pontszűkítés definíciója a következőképpen hangzik: egy C korlát *pontszűkítő*, ha minden olyan tár esetén tartományszűkítő, amelyben C változói legfeljebb egy kivétellel be vannak helyettesítve (másképpen: minden ilyen tár esetén a korlát a behelyettesítetlen változót pontosan a C reláció által megengedett értékekre szűkíti). Az egyetlen, nem pontszűkítő elemi korlát a `mod`.

Lineáris korlátok esetén a kifejtés megőrzi a pont-, illetve az intervallum-szűkítést, általános esetben azonban még a pont-szűkítést sem őrzi meg, pl:

```
| ?- X in 0..10, X*X*X*X #= 16.
X in 1..4 ? ;
no
```

A korlátkifejtés mechanizmusát a `goal_expansion/4` segédeljárás hívásával magunk is megvizsgálhatjuk:

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

```
        scalar_product([1,-2,-1],[Y,X,_A],#=,1)) ?

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

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

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

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

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

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

5.15. Segédeljárások a `clpfd`-ben

Statisztika

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

Példa:

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

```
    findall(Q, queens(Lab, N, Q), Ss),
    statistics(runtime, [_,Time]),
    fd_statistics(backtracks, Btrks).
```

A még le nem futott, alvó korlátok kiírása a válaszban)

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

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

5.16. FD-változók és FD-halmazok

5.16.1. definíció: *FD-változó*nak nevezünk minden olyan Prolog változót, amely a korlát-tár korlátaiban szerepel, és ezért a `clpfd` könyvtár különböző információkat tart fenn róla. Az FD-változókról tárolt információkat a programban felhasználhatjuk, de nagy valószínűséggel csak a címkézésnél, nyomkövetésnél, illetve globális korlátokban látjuk hasznukat.

Az FD-változókkal kapcsolatban a következő eljárásokat érdemes ismerni:

- `fd_var(V)`: `V` egy, a `clpfd` könyvtár által ismert változó.
- `fd_min(X, Min)`: a `Min` paramétert egyesíti az `X` változó tartományának alsó határával (ez egy szám vagy `inf` lehet).
- `fd_max(X, Max)`: a `Max` paramétert egyesíti az `X` változó tartományának felső határával (ez egy szám vagy `sup` lehet).
- `fd_size(X, Size)`: `Size` az `X` tartományának számossága (szám vagy `sup`).
- `fd_dom(X, Range)`: `Range` az `X` változó tartománya, tartománykifejezés formában
- `fd_set(X, Set)`: `Set` az `X` tartománya úgynevezett *FD-halmaz* formában (ld. lejjebb).
- `fd_degree(X, D)`: `D` az `X`-hez kapcsolódó korlátok száma.

Néhány példa a fentiek használatával:

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

A pár sorral előbb említett *FD-halmaz*ok a `clpfd` könyvtárban a tartományok belső ábrázolási formáját jelentik. Jelenleg egy FD-halmaz `[Alsó|Felső]` alakú szeparált zárt intervallumok rendezett listája (mivel a `.(_,_)` struktúra memóriaigénye 33%-kal kisebb, mint bármely más `f(_,_)` struktúráé), ezt azonban *szigorúan tilos* kihasználni, nem szabad ilyen adatszerkezetet a rendelkezésre álló könyvtári eljárásokon kívül más módszerrel létrehozni vagy megváltoztatni! Az ilyen jellegű megoldások előbb-utóbb `segmentation violation` vagy hasonló jellegű hibaüzenethez vezetnek.

Az adattípus manipulálására szolgáló könyvtári eljárások:

- `is_fdset(S)`: `S` egy korrekt FD-halmaz.
- `empty_fdset(S)`: `S` az üres FD-halmaz.
- `fdset_singleton(S,E)`: `S` egyedül az `E` elemet tartalmazza.
- `fdset_interval(Set, Min, Max)`: `S` egyedül a `Min..Max` intervallum elemeit tartalmazza
- `empty_interval(Min, Max)`: `Min..Max` egy üres intervallum (ekvivalens a `\+ fdset_interval(_, Min, Max)` hívással).
- `fdset_parts(S, Min, Max, Rest)`: Az `S` FD-halmaz áll egy `Min..Max` kezdő intervallumból és egy `Rest` maradék FD-halmazból, ahol `Rest` minden eleme nagyobb `Max+1`-nél. Egyaránt használható FD-halmaz szétszedésére és építésére.

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

- `fdset_union(Set1, Set2, Union)`: `Set1` és `Set2` uniója `Union`, `fdset_union(ListOfSets, Union)`: a `ListOfSets` FD halmazokból álló lista elemeinek uniója `Union`.
- `fdset_intersection/[3,2]` : Két halmaz, illetve egy listában megadott halmazok metszete (analóg az `fdset_union/[3,2]`-vel)
- `fdset_complement(Set1, Set2)`: `Set1` és `Set2` egymás komplemensei
- `fdset_member(Elt, Set)`: `Elt` eleme a `Set` FD-halmaznak. Visszalépésre az összes elemet felsorolja
- `list_to_fdset(List, Set)`, `fdset_to_list(Set, List)`: számlista átalakítása FD-halmazzá, és fordítva.
- `range_to_fdset(Range, Set)`, `fdset_to_range(Set, Range)`: tartománykifejezés átalakítása FD-halmazzá és viszont.

Itt jegyeznénk meg, hogy egy FD-halmaz szintén felhasználható egy változó tartományának szűkítésére az `X in_set Set` hívás használatával, azonban vegyük figyelembe, hogy ha `Set`-et egy `Y` változó tartományának függvényében alakítjuk ki, akkor ezzel még nem érünk el „démoni” hatást, mivel a szűkítés csak `Y` tartományának aktuális állásától függően fog végrehajtódni. Az `in_set` szűkítés leginkább globális korlátokban és testreszabott címkézésnél használatos.

5.17. A címkézés (labeling) testreszabása

Elevenítsük fel a CLP programok szerkezetére vonatkozóan tett megállapításainkat! Egy CLP programot három fő részre lehet szeparálni:

1. Változók felvétele és tartományaik megadása
2. Korlátok felvétele (lehetőség szerint választási pontok nélkül)
3. Címkézés, azaz a változók lehetséges értékeinek valamilyen rendszer szerint történő behelyettesítése

Ebben a fejezetben a 3. lépés lehetséges megvalósításaival fogunk foglalkozni. A címkézés során adott egy változóhalmaz, amelyben minden változónak egy lehetséges értékkészlete van. Egy *címkézési lépés*ben a következő teendőket végezzük el:

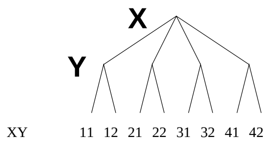
1. Kiválasztunk a címkézendő változók közül egyet
2. A kiválasztott változó értékkészletét olyan diszjunkt részhalmazokra bontjuk, amelyek egyesítése kiadja az eredeti értékkészletet (*particionálás*)
3. Ezen partíciókkal egy választási pontot hozunk létre, a választási pontból kivezető ágak mindegyike az adott változó értékkészletét az egyik partícióra szűkíti le. Az ágakat a hagyományos Prolog végrehajtás szabályai szerint járjuk be, figyelembe véve azt, hogy amikor egy ágat kiválasztunk, és a változó értékét az adott partícióra szűkítjük, akkor ez többnyire különböző korlátok felébredését okozza, amelyek meghiúsulást válthatnak ki. Ha a szűkítés sikeresen lefutott, akkor jöhet a következő címkézési lépés. (Megjegyzés: ha a kiválasztott változó értékkészlete még nem egyetlen számból áll, akkor semmi nem zárja ki azt, hogy a következő címkézési lépésben ugyanezt a változót válasszuk)

A keresés célja többféle lehet. Előfordulhat, hogy az *összes* megoldást meg szeretnénk keresni, előfordulhat, hogy egy *tetszőleges* megoldásra vagyunk kíváncsiak, és az is megeshet, hogy a megoldások közül valamilyen szempontból az *optimálisat* keressük. Arra azonban mindenképp törekednünk kell, hogy ez(eke)t a megoldás(oka)t a lehető legkevesebb visszalépés végrehajtásával, a lehető legrövidebb idő alatt találjuk meg. Ebben segít a keresési stratégia testreszabása. A testreszabás három szempont szerint történhet:

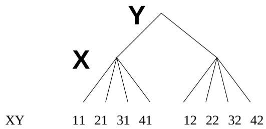
- A címkézési lépés elején a változó kiválasztásának módjával
- A választási pont fajtájával (kétszeres, többszörös), valamint az egyes ágakhoz hozzárendelt tartományokkal
- A választási pontok ágainak bejárási irányával

Nézzük meg egy egyszerű példán, hogy hogyan függ a keresési tér mérete a változó kiválasztásának módjától!

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




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

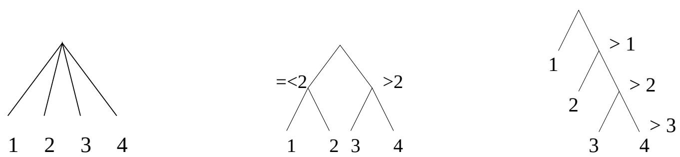


Ha feltételezzük, hogy a fenti keresés során egyes ágak meghiúsulhatnak, akkor könnyen rájöhetünk, hogy érdemesebb a második ábra szerinti keresési teret választani, mert ha itt a kezdő csomópontból kimenő egyik ág meghiúsul, akkor azzal már helyből 4 eset ellenőrzését spóroltuk meg. Ez az úgynevezett *first-fail* elv: előbb címkézzük a kisebb tartományú változót, ezzel remélhetőleg kevesebb választási pont lesz, csökken a keresési tér mérete. Az is előfordulhat, hogy egyes feladattípusokhoz saját, speciális sorrend a célszerű: például az N királynő feladatban célszerű előbb a középső sorokba elhelyezni a királynőket, mert ezek jobban megszűrik a maradék változók értékkészletét, mint a szélső sorokba helyezett királynők.

A `clpfd` könyvtár beépített címkéző eljárása, a `labeling/2` az első paraméterként átadott címkézési opció-listán keresztül lehetőséget ad arra, hogy a keresési tér szerkezetét befolyásoljuk. Három lehetőség közül választhatunk:

- *Felsorolás (`enum`)* — többszörös választási pontot hoz létre, pontosan annyi ággal, ahány lehetséges értéke van a változónak. Egy ágon mindig ezen értékek közül pontosan az egyikre történik behelyettesítés. Használatára példa:
  `| ?- X in 1..4, labeling([enum],[X]).`
- *Kettévágás (`bisect`)* — a változó tartományát megfelezi, és két ágat hoz létre, értelemszerűen a két résztartományra való behelyettesítésével. Használatára példa:
  `| ?- X in 1..4, labeling([bisect],[X]).`
- *Lépegetés (`step`)* — kiválaszt a változó tartományából egy értéket, és két ágat hoz létre. Az egyik ágon erre az értékre szűkíti a változó tartományát, a másik ágon pedig ezt az értéket kizárja a változó tartományából. Használatára példa:
  `| ?- X in 1..4, labeling([step],[X]).`

Az alábbi ábrákon egy 1..4 értékkészlettel rendelkező változó különböző címkézéseiből adódó keresési terek láthatóak.



Felsorolás (`enum`) Kettévágás (`bisect`) Lépegetés (`step`)

Ezen „rövid” bevezető után lássuk a `labeling/2` címkéző eljárás részletes ismertetését!

A címkézés alap-eljárása: `labeling(Opciók, VáltozóLista)`

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

1. a változó kiválasztása: *`leftmost`*`, min, max, ff, ffc, variable(Sel)`. Ezek jelentése:
   - `leftmost` — a változólista legbaloldalibb eleme.
   - `min` — a legkisebb alsó határú. Ha több van, akkor ezek közül a legbaloldalibb.
   - `max` — a legnagyobb felső határú. Ha több van, akkor ezek közül a legbaloldalibb.
   - `ff` — *first-fail* elv szerint a legkisebb tartományú (ld. `fd_size/2` az 5.16. fejezetben). Ha több van, akkor ezek közül a legbaloldalibb.
   - `ffc` — a legkisebb tartományú. Ha több ilyen van, akkor ezek közül az, amelyikhez a legtöbb korlát kapcsolódik (*most constrained* elv). Ha még mindig több ilyen van, akkor ezek közül a legbaloldalibb. Egy változóhoz kapcsolódó korlátok számát az `fd_degree/2` adja meg (ld. 5.16. fejezet).
   - `variable(Sel)` — testreszabott változó-kiválasztás: a következő változó kiválasztása a `Sel` felhasználói eljárás szerint történik. Bővebben erről a 68. oldalon még lesz szó.
2. a választási pont fajtája: ***`step`***`, enum, bisect, value(Enum)`. Ezek jelentése:
   - `step` — `X #= B` és `X #\= B` közti választás, ahol `B` az `X` tartományának alsó vagy felső határa, a bejárási iránytól függően
   - `enum` — többszörös választás `X` lehetséges értékei közül
   - `bisect` — `X #< M` és `X #>= M` közti választás, ahol `M` az `X` tartományának középső eleme.
   - `value(Enum)` — testreszabott választás: `Enum` egy felhasználói eljárás, amelynek szerepe, hogy leszűkítse X tartományát. Bővebben erről az 69. oldalon még lesz szó.
3. a bejárási irány: ***`up`***`, down`. `up` értelemszerűen alulról felfelé, `down` felülről lefelé járja be a tartományt. Csak `step` típusú címkézésnél van szerepe.
4. a keresett megoldások: ***`all`***`, minimize(X), maximize(X)`. Ezek jelentése:
   - `all` – az összes megoldást megkeresi
   - `minimize(X)` – azt a megoldást adja vissza, melyben `X` értéke minimális.
   - `maximize(X)` – azt a megoldást adja vissza, melyben `X` értéke maximális.
5. a gyűjtendő statisztikai adat: `assumptions(A)`. A keresés végén egyesíti `A` értékét a sikeres megoldáshoz vezető ágon lévő változó-kiválasztások számával (ami lényegében a keresési út hosszát jelenti).
6. a balszélső ágtól való eltérés korlátozása: `discrepancy(D)`. Ezzel azt korlátozzuk, hogy a keresés során maximum `D`-szer választhatunk a választási pontokban nem legbaloldalibb ágat. Akkor hasznos, ha a probléma megoldására van valamiféle heurisztikánk, és úgy alakítjuk ki a keresési teret, hogy a heurisztika szerinti optimális választást a legbaloldalibb ág tartalmazza. Mivel a heurisztika nem teljesen tökéletes, ezért valamekkora eltérést megengedünk (ezt szabályozzuk `D` értékével). Ezt a módszert hívjuk *Limited Discrepancy Search*-nek (LDS).

A fenti pontokra való hivatkozással a `labeling/2` eljárás pontos működése így fest:

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

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

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

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

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

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

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

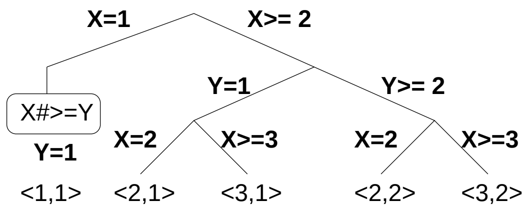
h. Eközben értelemszerűen követjük a 4-6. csoportbeli opciók előírásait is.

Lássunk egy példát az `fdbg` nyomkövető könyvtár (6. fejezet) használatával!

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

```
    Labeling [6, <y>]: failed.
Labeling [1, <x>]: failed.
```

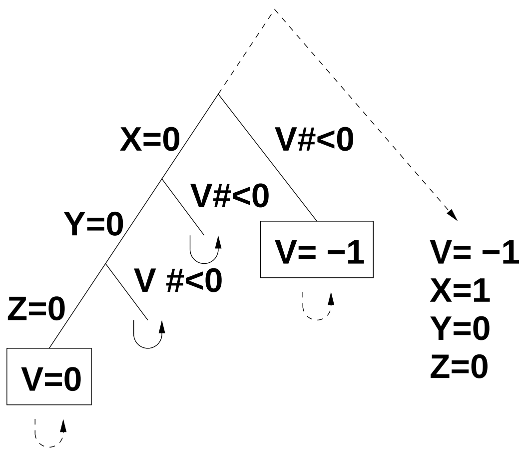
A keresési fa:



Egy másik példa, ezúttal szélsőérték-számításra:

```
| ?- _L=[X,Y,Z], domain(_L, 0, 1), V#=Y+Z-X, labeling([minimize(V)], _L).
V = -1, X = 1, Y = 0, Z = 0 ? ;
no
```

A keresési fa (branch-and-bound algoritmussal):



A branch-and-bound algoritmus itt először megkeresi az első megoldást, majd innentől kezdve a további ágak végigjárása során korlátként felveszi azt is, hogy a megoldásnak kisebbnek kell lennie, mint az eddig megtalált legkisebb. Ha ezzel a feltétellel is talál újabb megoldást, akkor innentől kezdve a többi ágon már ezzel az újabb minimummal dolgozik, egészen addig, amíg be nem járja a teljes keresési teret.

A statisztikai funkciót és a bal szélső ágtól való eltérés korlátozását bemutató példa (érdemes összevetni az eredményeket a 64. oldalon látható keresési fákkal):

```
% a Select címkézési mód használatával megkeresi az X in 1..4 korlát összes
% megoldását, és a megoldásokhoz vezető utak hosszát As-ben adja vissza
assumptions(Select, As) :-
     X in 1..4, findall(A, labeling([Select, assumptions(A)], [X]), As).

% a Select címkézési mód és D eltérés-korlát használatával megkeresi az
```

```
% X in 1..4 korlát összes megoldását, és a megtalált megoldásokat visszaadja
% Xs-ben
lds(Select, D, Xs) :-
     X in 1..4, findall(X, labeling([Select, discrepancy(D)], [X]), Xs).

| ?- assumptions(enum, As).          As = [1,1,1,1] ? ; no
| ?- assumptions(bisect, As).        As = [2,2,2,2] ? ; no
| ?- assumptions(step, As).          As = [1,2,3,3] ? ; no

| ?- lds(enum, 1, Xs).               Xs = [1,2,3,4] ? ; no
| ?- lds(bisect, 1, Xs).             Xs = [1,2,3] ? ; no
| ?- lds(step, 1, Xs).               Xs = [1,2] ? ; no
```

Látható, hogy `enum` címkézési módnál minden megoldáshoz 1 hosszú út vezet, mivel egy többszörös választási pontot hoztunk létre. Éppen ezért az a korlátozásunk, hogy a legbaloldalibb ágtól maximum egyszer térhetünk el, nem jelent semmi pluszt, mind a négy megoldást meg tudjuk így keresni.

`bisect` címkézésnél minden megoldáshoz egy 2 hosszú út vezet, hiszen az első választási pontban az `X #< 3` és `X #>= 3` korlátok felvétele között választunk, a második választási pontban pedig a megmaradó 2 méretű tartományokat felezzük tovább. A legbaloldalibb ágtól való eltérésre vonatkozó korlátunk így nem találja meg a 4-et, mint megoldást, mert ahhoz először az `X #>= 3` ágat, másodszor pedig az `X = 4` ágat kéne választanunk, és mindkettő jobb oldali ág.

`step` címkézésnél az 1-hez, mint megoldáshoz 1 hosszú út vezet, mert az első választási pontban az `X = 1` és `X #= 1` korlátok között döntünk. A 2-t így már csak két lépésben tudjuk megtalálni, a 3-at és a 4-et pedig hasonló módon csak 3-3 lépésben. Mivel a bal oldali út itt mindig az `X` valamely értékre való kötését jelenti, és csak egyszer léphetünk jobb oldalra, ezért csak az 1-et és a 2-t fogjuk megtalálni, hiszen a 3-hoz már kétszer, a 4-hez már háromszor kéne jobbra lépnünk.

Mint azt már néhány oldallal előbb említettük, a felhasználónak a `labeling/2` eljárás `variable(Sel)` opcióján keresztül lehetősége van egy saját változó-kiválasztó eljárás írására is. Az eljárást `Sel(Vars, Selected, Rest)` alakban hívja meg a rendszer, ahol `Vars` a még címkézendő változók/számok listája. `Sel` feladata, hogy determinisztikusan egyesítse `Selected`-et a következő címkézendő *változóval*, `Rest`-et pedig a maradékkal. `Sel` egy tetszőleges meghívható kifejezés lehet, a három argumentumot a rendszer fűzi `Sel` argumentumainak végére, így a `Sel` által hivatkozott eljárás akár 3-nál több paraméterrel is rendelkezhet. A példa egy olyan kiválasztást valósít meg, ahol a felhasználó szabályozhatja, hogy hányadik változót válasszuk ki a változó-listából.

```
% A Vars-beli változók között Sel a Hol-adik, ha a lista teljes hosszát
% 1-nek vesszük (Hol így egy törtszám). Rest a maradék.
valaszt(Hol, Vars, Sel, Rest) :-
        szur(Vars, Szurtek),
        length(Szurtek, Len), N is integer(Hol*Len),
        nth0(N, Szurtek, Sel, Rest).

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

Lehetőség van a kiválasztott változó tartományának szűkülését is befolyásolni. Ehhez a `labeling/2` opció-listájában a `value(Enum)` opciót kell megadnunk. `Enum`-ot a rendszer `Enum(X, Rest, BB0, BB)` alakban hívja meg, ahol `[X|Rest]` a címkézendő változók listája, és ebből `X`-et kell az eljárásnak címkéznie, mégpedig nemdeterminisztikus módon `X` tartományát az összes kívánt módon szűkítve (tehát a `value(Enum)` a 66. oldalon lévő lépések közül a **c.**, **d.** és **e.** lépéseket váltja ki). `BB` és `BB0` értéke számunkra csak annyiból lényeges, hogy az első választásnál meg kell hívni `first_bound(BB0, BB)`-t, a másodiknál pedig `later_bound(BB0, BB)`-t a branch-and-bound, illetve LDS módszerek kiszolgálására. `Enum` egy meghívható kifejezés, a 4 paramétert a rendszer fűzi `Enum` argumentumlistájának végére. Példaként tekintsünk egy olyan címkéző eljárást, amely az értékeket az értéklistában belülről kifelé haladva sorolja fel:

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

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

Végül, a címkézés testreszabásának fontosságát bizonyítandó, nézzük meg az N királynő feladat megoldását különféle címkéző módszerekkel (600 MHz-es Pentium III gépen):

Összes megoldás keresése

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

Első megoldás keresése

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

5.18. Kombinatorikus korlátok

Az ebben a fejezetben ismertetett globális korlátok az eddigiekhez hasonlóan nem tükrözhetőek. Minden olyan helyen, ahol a korlátok FD-változót várnak, írhatunk számértéket is.

5.18.1. Értékek számolása és különbözősége

`count(Val, List, `*`Relop`*`, Count)`

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

`global_cardinality(Vars, Vals)`

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

`all_different(Vs`*`[`*`, Options`*`]`*`)`
`all_distinct(Vs`*`[`*`, Options`*`]`*`)`

Jelentése: a `Vs` FD változó-lista elemei páronként különbözőek. A korlát szűkítési mechanizmusát az `Options` opció-lista szabályozza, eleme lehet:

- `consistency(Cons)` — a szűkítési algoritmust szabályozza. `Cons` lehet:

  `global` — tartomány-szűkítő algoritmus (Regin), durván a tartományok méretével arányos idejű (alapértelmezés `all_distinct` esetén)

  `bound` — intervallum-szűkítő algoritmus (Mehlhorn), a változók és értékek számával arányos idejű

  `local` — a nemegyenlőség páronkénti felvételével azonos szűkítő erejű algoritmus, durván a változók számával arányos idejű (alapértelmezés `all_different` esetén).

- `on(On)` — az ébredést szabályozza. `On` lehet:

[1] *`midvar`* ≡ `variable(valaszt(0.5))`.

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

dom — a változó tartományának bármiféle változásakor ébreszt (alapértelmezés `all_distinct` esetén),

min, max, ill. **minmax** — a változó tartományának adott ill. bármely határán történő változáskor ébreszt,

val — a változó behelyettesítésekor ébreszt csak (alapértelmezés `all_different` esetén).

A `consistency(local)` beállításnál nincs értelme `val`-nál korábban ébreszteni, mert ez a szűkítést nem befolyásolja.

A különböző ébresztési és szűkítési módok bemutatásához nézzük az alábbi predikátumot:

```
pelda(Z, I, On, C) :-
     L = [X,Y,Z], domain(L, 1, 3),
     all_different(L, [on(On),consistency(C)]), X #\= I, Y #\= I.
```

```
| ?- pelda(Z, 3, dom, local).       →   Z in 1..3
| ?- pelda(Z, 3, min, global).      →   Z in 1..3
| ?- pelda(Z, 3, max, bound).       →   Z = 3
| ?- pelda(Z, 2, minmax, global).   →   Z in 1..3
| ?- pelda(Z, 2, dom, bound).       →   Z in 1..3
| ?- pelda(Z, 2, dom, global).      →   Z = 2
```

Látható, hogy csak a harmadik és a hatodik példa jön rá arra, hogy `Z` csak a második paraméterben megadott érték lehet. Az első és az ötödik példa a `local`, illetve `bound` algoritmus gyengesége miatt nem végzi el a szűkítést, a második és a negyedik pedig a nem megfelelő ébresztési feltétel miatt.

5.18.2. Függvénykapcsolatok és relációk

`element(X, List, Y)`

Jelentése: `List` `X`-edik eleme `Y` (1-től számozva). Itt `X` és `Y` FD változók, `List` FD változókból álló lista. Az `X` változóra nézve tartomány-szűkítést, az `Y` és `List` változókra nézve intervallum-szűkítést biztosít.

Példák:

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

% X #= C #<=> B megvalósítása, 1=<{X,C}=<6 esetére
% (C konstans).
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, Rel, Y)`

Itt `X` és `Y` FD változók, `Rel` formája: egy lista *Egész-KonstansTartomány* alakú párokból (ahol mindegyik *Egész* csak egyszer fordulhat elő). Jelentése: `Rel` tartalmaz egy `X-Tart` párt, ahol `Y` eleme a `Tart`-nak, azaz:

$$\texttt{relation(X,H,Y)} \equiv \langle \texttt{X},\texttt{Y}\rangle \in \{\langle x,y\rangle \,|\, x-\texttt{T}\in \texttt{H}, y\in \texttt{T}\}$$

Tetszőleges bináris reláció definiálására használható, tartomány-szűkítést biztosít. Példa:

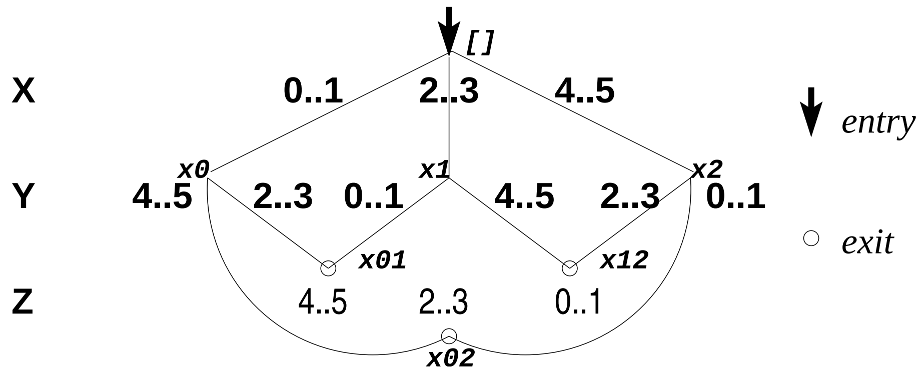
```
'abs(x-y)>1'(X,Y) :-
        relation(X, [0-(2..5), 1-(3..5), 2-{0,4,5},
                     3-{0,1,5}, 4-(0..2), 5-(0..3)], Y).

| ?- 'abs(x-y)>1'(X,Y), X in 2..3.
Y in (0..1) \/ (4..5) ? ;
no
```

case(Template, Tuples, DAG*[*, Options*]*)

Jelentése: A `Tuples` lista minden elemét illesztve a `Template` mintára, a `DAG` által leírt reláció fennáll. Az ébresztést és a szűkítést az `Options` opció-lista szabályozza (hasonló módon, mint az `all_distinct` esetén, lásd 70. oldal és SICStus kézikönyv). Alaphelyzetben minden változásra ébred és tartomány-szűkítést ad.

A `DAG` csomópontok listája, az első elem a kezdőpont. Egy csomópont alakja: `node(ID, X, Successors)`. Itt `ID` a csomópont azonosítója (egész vagy atom), `X` a vizsgálandó változó. Belső gráfpont esetén `Successors` a rákövetkező csomópontok listája, elemei `Min..Max)-ID2` alakúak. Egy ilyen elem jelentése: ha `Min` $\leq$ `X` $\leq$ `Max`, akkor menjünk az `ID2` csomópontra. Végpont esetén `Successors` a végfeltételek listája, elemei `Min..Max` alakúak, jelentése pedig: ha valamelyik elem esetén `Min` $\leq$ `X` $\leq$ `Max` fennáll, akkor a reláció teljesül.

Példaként tekintsük az alábbi gráfot, amely az „`X`, `Y` és `Z` felének egészrésze mind más” ($[\frac{X}{2}] \neq [\frac{Y}{2}], [\frac{X}{2}] \neq [\frac{Z}{2}], [\frac{Y}{2}] \neq [\frac{Z}{2}]$) relációt írja le a 0..5 tartományon:



Ennek megvalósítása a `case/3` korláttal:

```
felemasok(X, Y, Z) :-
   case(f(A,B,C), [f(X,Y,Z)],
        [node([], A, [(0..1)-x0,(2..3)-x1,(4..5)-x2]),
         node(x0, B, [(2..3)-x01,(4..5)-x02]),
         node(x1, B, [(0..1)-x01,(4..5)-x12]),
         node(x2, B, [(0..1)-x02,(2..3)-x12]),
         node(x01,C,[4..5]), node(x02,C,[2..3]), node(x12,C,[0..1])
        ]).
```

Példa többszörös mintára: `case(T,[`A_1`,...],D)`$\equiv$`case(T,[`A_1`],D),...`

```
felemasok_vacak(X, Y, Z) :-
    case(A\=B, [X\=Y,X\=Z,Y\=Z],
     [node(root, A, [(0..1)-0,(2..3)-1,(4..5)-2]),
      node(0,B,[2..5]),node(1,B,[0..1,4..5]),node(2, B, [0..3])
     ], [on(val(X)),on(val(Y)),on(val(Y))/*,prune(val(X)), ...*/]).
```

5.18.3. Leképezések, gráfok

`sorting(X, I, Y)`

Az `X` FD-változókból álló lista rendezettje az `Y` FD-változó-lista. Az `I` FD-változó-lista írja le a rendezéshez szükséges permutációt. Azaz: mindhárom paraméter azonos (n) hosszúságú, `Y` rendezett, `I` az `1..`n számok egy permutációja, és $\forall i \in$ `1..`n esetén $\mathtt{X}_i = \mathtt{Y}_{\mathtt{I}_i}$.

`assignment(X, Y[, Options])`

`X` és `Y` FD változókból alkotott azonos (n) hosszúságú listák. Teljesül, ha $\mathtt{X}_i$ és $\mathtt{Y}_i$ mind az `1..`n tartományban vannak és $\mathtt{X}_i = j \Leftrightarrow \mathtt{Y}_j = i$. Másképpen fogalmazva: `X` egy-egyértelmű leképezés az `1..`n halmazon (az `1..`n számok egy permutációja), és `Y` az `X` inverze.

Az `Options` lista ugyanolyan, mint az `all_different/[1,2]` korlát esetében (ld. 70. oldal), az alapértelmezés `[on(domain),consistency(global)]`.

`circuit(X)`

`X` n hosszúságú lista. Igaz, ha minden $\mathtt{X}_i$ az `1..`n tartományba esik, és $\mathtt{X}_1$, $\mathtt{X}_{\mathtt{X}_1}$, $\mathtt{X}_{\mathtt{X}_{\mathtt{X}_1}}$ ... (n-szer ismételve) az `1..`n számok egy permutációja. Másképp: `X` egy egyetlen ciklusból álló permutációja az `1..`n számoknak.

Gráfokon vett értelmezés: legyen egy n szögpontú irányított gráfunk, jelöljük a pontokat az `1..`n számokkal. Vegyünk fel n db FD-változót, $\mathtt{X}_i$ tartománya álljon azon j számokból, amelyekre i-ből vezet j-be él. Ekkor `circuit(X)` azt jelenti, hogy az $i \rightarrow \mathtt{X}_i$ élek a gráf egy Hamilton-körét adják.

`circuit(X, Y)`

Ekvivalens a következővel: `circuit(X), assignment(X, Y)`. Gráfokon értelmezve megadja a Hamilton-kört és annak az ellenkező irányban vett bejárását is.

Példák az `assignment/2` és a `circuit/2` használatára:

```
| ?- length(L, 3), domain(L, 1, 3), assignment(L, LInv), L=[2|_],
     labeling([], L).
L = [2,1,3], LInv = [2,1,3] ? ;
L = [2,3,1], LInv = [3,1,2] ? ;
no
| ?- length(L, 3), domain(L, 1, 3), circuit(L, LInv), L=[2|_].
L = [2,3,1], LInv = [3,1,2] ? ;
no
```

Kicsit „életszagúbb” példa:

1	2	2
3	1	3
4	4	1

Adott a bal oldalt látható 3 × 3-as négyzetrács. Feladat: járjuk be a rács elemeit a bal felső sarokból indulva úgy, hogy minden cellán pontosan egyszer haladunk át, és n-t tartalmazó celláról csak $n+1$-et tartalmazóra léphetünk (kivéve $n=4$ esetben, innen csak 1-est tartalmazóra).

```
| ?- L=[_1,_2,_3,_4,_5,_6,_7,_8,1], _1=2, _2 in {4,6}, _3=6,
     _4 in {7,8}, _5 in {2,3}, _6=8, _7=5, _8 in {5,9},
```

```
     circuit(L).
L = [2,4,6,7,3,8,5,9,1] ? ; no
```

Az eredmény-listában minden elem megadja, hogy az elemnek megfelelő cella után melyik cella következik a körben (a cellákat fentről lefelé és balról jobbra számoztuk be 1-től 9-ig).

A `circuit/1` felhasználható az utazó ügynök probléma megoldására is:

```
:- 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] ?
```

5.18.4. Ütemezési korlátok

`cumulative(Starts, Durations, Resources, Limit[, Opts])`

Jelentése: a `Starts` kezdőidőpontokban elkezdett, `Durations` ideig tartó és `Resources` erőforrásigényű feladatok bármely időpontban összesített erőforrásigénye nem haladja meg a `Limit` határt (és fennállnak az opcionális precedencia korlátok). Az első három argumentum FD változókból álló, egyforma (n) hosszú lista, a negyedik egy FD változó.

`serialized(Starts, Durations[, Opts])`

A `cumulative` speciális esete, ahol az összes erőforrás-igény és a korlát is 1.

Vezessük be a `cumulative(`S, D, R, Lim `…)` híváshoz az alábbi jelöléseket:

$a = min(S_1, \dots, S_n)$ a a kezdőidőpont)
$b = max(S_1 + D_1, \dots, S_n + D_n)$ b a (végidőpont)
$R_{ij} = R_j$, ha $S_j \le i < S_j + D_j$, egyébként $R_{ij} = 0$; R_{ij} (a j. feladat erőforrásigénye az i. időpontban)

Ezekkel a jelölésekkel a korlát jelentése (a precedencia-korlátok nélkül):

$R_{i1} + \dots + R_{in} \le Lim$ minden $a \le i < b$ esetén.

Az `Opts` opciólista a következő beállításokat tartalmazhatja:

- `precedences(Ps)` — `Ps` egy lista, amely precedencia korlátokat ír le. Elemei a következők lehetnek (`I` és `J` feladatok sorszámai, `D` egy pozitív egész, `Tart` egy konstans-tartomány):
  - `d(I,J,D)`, jelentése: $S_{\mathtt{I}} + \mathtt{D} \le S_{\mathtt{J}}$ vagy $S_{\mathtt{J}} \le S_{\mathtt{I}}$ (tehát az `I.` és a `J.` feladat közti átállás egy holtidővel modellezhető, és `D` az `I.` feladat hossza ($D_{\mathtt{I}}$) plusz az átállási idő. Ha azt akarjuk megadni, hogy egy feladatot előbb el kell végezni, mint egy másikat, akkor átállási időnek `sup`-ot kell megadni.
  - `I-J in Tart`, jelentése: $S_{\mathtt{I}} - S_{\mathtt{J}}$ `#=` $\mathtt{D}_{\mathtt{IJ}}$, $\mathtt{D}_{\mathtt{IJ}}$ `in Tart`. Akkor használatos, ha az `I.` és `J.` feladat között eltelt időnek alsó és felső korlátja is van.
- `resource(R)` — speciális ütemezési címkézéshez szükséges opció. `R`-et egyesíti egy kifejezéssel, amelyet később átadhatunk az `order_resource/2` eljárásnak, hogy felsoroltassuk a feladatok lehetséges sorrendjeit. Az `order_resource/2` eljárásról bővebben a 77. oldalon lesz szó.
- `decomposition(Boolean)` — Ha `Boolean true`, akkor minden ébredéskor megpróbálja kisebb darabokra bontani a korlátot (pl. ha van két át nem lapoló feladathalmazunk, akkor ezeket külön-külön kezelhetjük, ami az algoritmusok gyorsabb lefutását eredményezheti). Alapértelmezésben ki van kapcsolva.
- `path_consistency(Boolean)` — Ha `Boolean true`, akkor figyeli a feladatok kezdési időpontja közti különbségek konzisztenciáját. Ez egy olyan redundáns korlátra hasonlít, amely minden i, j párra felveszi a $\mathtt{SD}_{ij}$ `#=` $\mathtt{S}_j$ `-` $\mathtt{S}_i$, és minden i, j, k hármasra a $\mathtt{SD}_{ik}$ `#=` $\mathtt{SD}_{ij}$ `+` $\mathtt{SD}_{jk}$ korlátot. Alapértelmezésben ki van kapcsolva.
- `static_sets(Boolean)` Ha `Boolean true`, akkor, ha bizonyos feladatok sorrendje ismert, akkor ennek megfelelően megszorítja azok kezdő időpontjait. Alapértelmezésben ki van kapcsolva. Például:

```
| ?- _S = [S1,S2,S3],  domain(_S, 0, 9), (SS = false ; SS = true),
     serialized(_S, [5,2,7], [static_sets(SS),
                              precedences([d(3,1,sup), d(3,2,sup)])]).
SS=false, S1 in 0..4, S2 in(0..2)\/(5..7), S3 in 5..9 ? ;
SS=true,  S1 in 0..4, S2 in(0..2)\/(5..7), S3 in 7..9 ? ;
no
```

- `edge_finder(Boolean)` Ha `Boolean` `true`, akkor megpróbálja kikövetkeztetni az egyes feladatok sorrendjét. Alapértelmezésben ki van kapcsolva. Példa:

```
| ?- _S = [S1,S2,S3], domain(_S, 0, 9),
       serialized(_S, [8,2,2], [edge_finder(true)]).
S1 in 4..9, S2 in 0..7, S3 in 0..7 ? ;
no
```

- `bounds_only(Boolean)` Ha `Boolean` `true`, akkor a korlát az S_i változóknak csak a határait szűkíti, a belsejüket nem. Alapértelmezésben be van kapcsolva.

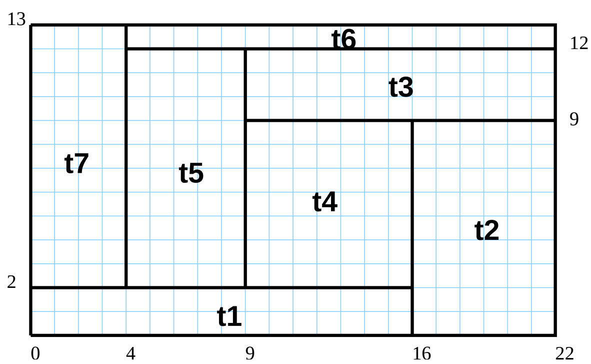
`cumulatives(Tasks, Machines[, Options])`
Több erőforrást (gépet) igénylő feladatok ütemezése (lásd SICStus kézikönyv).

Példaként tekintsünk egy olyan ütemezési feladatot, ahol a rendelkezésre álló erőforrások száma 13, az erőforrásigények és időtartamok pedig az alábbi táblázat szerint alakulnak:

Tevékenység	t1	t2	t3	t4	t5	t6	t7
Időtartam	16	6	13	7	5	18	4
Erőforrásigény	2	9	3	7	10	1	11
Egy megoldás	0–16	16–22	9–22	9–16	4–9	4–22	0–4

```
% A fenti ütemezési feladatban a tevékenységek kezdőidőpontjait
% az Ss lista tartalmazza, a legkorábbi végidőpont az End.
schedule(Ss, End) :-
        length(Ss, 7),
        Ds = [16, 6,13, 7, 5,18, 4],
        Rs = [ 2, 9, 3, 7,10, 1,11],
        domain(Ss, 0, 30),
        End in 0.. 50,
        after(Ss, Ds, End),
        cumulative(Ss, Ds, Rs, 13),
        labeling([ff,minimize(End)], [End|Ss]).

% after(Ss, Ds, E): Az E időpont az Ss kezdetű Ds időtartamú
% tevékenységek mindegyikének befejezése után van.
after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).
```

```
| ?- schedule(Ss, End).

Ss = [0,16,9,9,4,4,0],
End = 22 ? ;
no
```



Példa precedencia-korlátra:

```
| ?- _S = [S1,S2], domain(_S,0,9), S1 #< S2, % a két külön korlát
     serialized(_S, [4,4], []).             % nem jól szűkít:
        S1 in 0..8, S2 in 1..9 ? ;  no

| ?- _S = [S1,S2], domain(_S,0,9), Opts=[precedences([d(2,1,sup)],
     serialized(_S, [4,4], Opts)]). % ^^ ≡ S1 #< S2
        S1 in 0..5, S2 in 4..9 ? ;  no
```

`order_resource(Options, Resource)`

Igaz, ha a `Resource` által leírt feladatok elrendezhetők valamilyen sorrendbe. Ezeket az elrendezéseket felsorolja. A `Resource` argumentumot a fenti ütemező eljárásoktól kaphatjuk meg az ütemező eljárás opció-listájába helyezett `resource(Resource)` elemmel. Az `order_resource/2` opció-listája a következő dolgokat tartalmazhatja (mindegyik csoportból legfeljebb egyet, alapértelmezés: `[first,est]`):

- stratégia
  - `first` Mindig olyan feladatot választunk ki, amelyet az összes többi elé helyezhetünk.
  - `last` Mindig olyan feladatot választunk ki, amelyet az összes többi után helyezhetünk.
- tulajdonság: `first` stratégia esetén az adott tulajdonság minimumát, `last` esetén a maximumát tekintjük az összes feladatra nézve.
  - `est` legkorábbi lehetséges kezdési idő
  - `lst` legkésőbbi lehetséges kezdési idő
  - `ect` legkorábbi lehetséges befejezési idő
  - `lct` legkésőbbi lehetséges befejezési idő

Példa:

```
| ?- _S=[S1,S2,S3], domain(_S, 0, 9),
     serialized(_S, [5,2,7],
                [precedences([d(3,1,sup), d(3,2,sup)]),
                 resource(_R)]), order_resource([],_R).
S1 in 0..2, S2 in 5..7, S3 in 7..9 ? ;
S1 in 2..4, S2 in 0..2, S3 in 7..9 ? ;
no
```

Látható, hogy az `order_resource/2` csak a lehetséges elrendezésekre vonatkozóan címkéz, de az egyes elrendezéseken belül a változók értékeit „függőben” hagyja.

5.18.5. Diszjunkt szakaszok és téglalapok

`disjoint1(Lines[, Options])`

Jelentése: A `Lines` által megadott intervallumok diszjunktak. A `Lines` lista elemei $F(S_j, D_j)$ vagy $F(S_j, D_j, T_j)$ alakú kifejezések listája, ahol S_j és D_j j. szakasz kezdőpontját és hosszát megadó változók. F tetszőleges funktor, T_j egy atom vagy egy egész, amely a szakasz típusát definiálja (alapértelmezése 0). Az `Options` lista a következő dolgokat tartalmazhatja (a `Boolean` változók alapértelmezése `false`):

- `decomposition(Boolean)` Ha `Boolean` `true`, akkor minden ébredéskor megpróbálja kisebb darabokra bontatni a korlátot.
- `global(Boolean)` Ha `Boolean` `true`, akkor egy redundáns algoritmust használ a jobb szűkítés érdekében. Példa:

```
| ?- domain([S1,S2,S3], 0, 9), (G = false ; G = true),
     disjoint1([S1-8,S2-2,S3-2], [global(G)]).
       G = false, S1 in 0..9, S2 in 0..9, S3 in 0..9 ? ;
       G = true,  S1 in 4..9, S2 in 0..7, S3 in 0..7 ?
```

- `wrap(Min,Max)` A szakaszok nem egy egyenesen, hanem egy körön helyezkednek el, ahol a `Min` és `Max` pozíciók egybeesnek (`Min` and `Max` egészek kell, hogy legyenek). Ez az opció a `Min..(Max-1)` intervallumba kényszeríti a kezdőpontokat.
- `margin(T1,T2,D)` Bármely `T1` típusú vonal végpontja legalább `D` távolságra lesz bármely `T2` típusú vonal kezdőpontjától, ha `D` egész. Ha `D` nem egész, akkor a `sup` atomnak kell lennie, ekkor minden `T2` típusú vonalnak előrébb kell lennie, mint bármely `T1` típusú vonal.

`disjoint2(Rectangles[, Options])`

Jelentése: A `Rectangles` által megadott téglalapok nem metszik egymást. A `Rectangles` lista elemei $F(S_{j1}, D_{j1}, S_{j2}, D_{j2})$ vagy $F(S_{j1}, D_{j1}, S_{j2}, D_{j2}, T_j)$ alakú kifejezések. Itt S_{j1} és D_{j1} a j. téglalap X irányú kezdőpontját és hosszát jelölő változók, S_{j2} és D_{j2} ezek Y irányú megfelelői, F tetszőleges funktor, T_j egy egész vagy atom, amely a téglalap típusát jelöli (alapértelmezése 0).

Az `Options` lista a következő dolgokat tartalmazhatja (a `Boolean` változók alapértelmezése `false`):

- `decomposition(Boolean)` Mint `disjoint1/2`.
- `global(Boolean)` Mint `disjoint1/2`.
- `wrap(Min1,Max1,Min2,Max2)` `Min1` és `Max1` egész számok vagy rendre az `inf` vagy `sup` atom. Ha egészek, akkor a téglalapok egy olyan henger palástján helyezkednek el, amely az X irányban fordul körbe, ahol a `Min1` és `Max1` pozíciók egybeesnek. Ez az opció a `Min1..(Max1-1)` intervallumba kényszeríti az S_{j1} változókat. `Min2` és `Max2` ugyanezt jelenti Y irányban. Ha mind a négy paraméter egész, akkor a téglalapok egy tóruszon helyezkednek el.
- `margin(T1,T2,D1,D2)` Ez az opció minimális távolságokat ad meg, `D1` az X, `D2` az Y irányban bármely `T1` típusú téglalap vég- és bármely `T2` típusú téglalap kezdőpontja között. `D1` és `D2` egészek vagy a `sup` atom. `sup` azt jelenti, hogy a `T2` típusú téglalapokat a `T1` típusú téglalapok elé kell helyezni a megfelelő irányban.
- `synchronization(Boolean)`: Speciális esetben redundáns korlátot vesz fel (lásd SICStus kézikönyv).

Példa: helyezzünk el három diszjunkt téglalapot úgy, hogy (x, y) bal alsó sarkuk az $0 \leq x \leq 2, 0 \leq y \leq 1$ téglalapban legyen. A méretek ($x \times y$ sorrendben): 1×3, 2×2, 3×3. Az 1×3-as téglalap x koordinátája nem lehet 2.

```
| ?- domain([X1,X2,X3], 0, 2), domain([Y1,Y2,Y3], 0, 1), X1 #\= 2,
     disjoint2([r(X1,3,Y1,1),r(X2,2,Y2,2),r(X3,3,0,3)]).
X1 in 0..1, Y1 = 0,    X2 = 0, Y2 = 1,    X3 = 2, Y3 = 1 ?
```

5.19. Felhasználói korlátok definiálása

A SICStus Prolog kétféle lehetőséget kínál a `clpfd` modul korlátainak felhasználói korlátokkal való bővítésére: a *globális korlát*okat és az *FD predikátum*okat. A *globális korlát*ok tetszőleges (nem korlátos) számú változót tartalmazó korlátok definiálására alkalmasak. A korlátok működését teljesen általános Prolog kódként adhatjuk meg, beleértve az ébresztési feltételeket és a befejezés módját is. A globális korlátok reifikációja (tükrözése) azonban nem támogatott. Az *FD predikátumok* ezzel szemben csak rögzített számú változót tartalmazó korlátok leírására alkalmasak, viszont itt a reifikáció is támogatott, és az ébresztési feltételek meghatározása automatikus. Az FD predikátumokban a programozó úgynevezett *indexikálisok* segítségével írja le a szűkítési, illetve levezethetőségi feltételeket. Az indexikálisok nyelve egy speciális, halmazértékű funkcionális nyelv a tartományokkal való műveletek végzésére. Az alábbi Prolog kód egy példa egy FD predikátumra:

```
% Az X+Y #= T korlát (intervallum szűkítéssel)
'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).
```

Az indexikális nyelv bővebb elemzése az 5.19.2. fejezetben olvasható.

5.19.1. Globális korlátok

Mint azt már említettük, a globális korlátot egy külön Prolog eljárásként kell megírni, amelyben az `fd_global/3` eljárással indul el a korlát tényleges végrehajtása. Az `fd_global/3` paraméterezése:

`fd_global(Constraint, State, Susp)`
Elindítja `Constraint` végrehajtását `State` kezdőállapottal és `Susp` ébresztési feltételekkel. `Constraint` egy tetszőleges Prolog struktúra lehet, azonban célszerű a korlát nevével megegyezőre választani, már csak azért is, mert ha a `clpfd:full_answer` bekapcsolásával kérjük a le nem futott démonok megjelenítését, akkor a Prolog a `Constraint`-ben megadott nevet fogja kiírni.

A Prolog lehetőséget biztosít arra, hogy a globális korlát az ébresztések között megőrizzen bizonyos állapotinformációkat. Ez az állapotinformáció is tetszőleges Prolog struktúra lehet, a kezdőértékét pedig a `State` paraméterrel tudjuk beállítani.

A korlát indításakor az `fd_global/3` harmadik paraméterében meg kell adni egy ébresztési listát, amely előírja, hogy mely változók milyen tartomány-változásakor kell felébreszteni a korlátot. A lista elemei a következők lehetnek:

- `dom(X)` — az `X` változó tartományának bármely változásakor
- `min(X)` — az `X` változó alsó határának változásakor
- `max(X)` — az `X` változó felső határának változásakor
- `minmax(X)` — az `X` változó alsó vagy felső határának változásakor
- `val(X)` — az `X` változó behelyettesítésekor

Fontos, hogy a korlát *nem* tudja majd, hogy melyik változójának milyen változása miatt ébresztik fel. Ráadásul ha több változás történik, a korlát akkor is csak egyszer fog felébredni, éppen ezért nagyon fontos, hogy a korlát minden lehetséges tartomány-változásra megfelelően reagáljon anélkül, hogy tudná, hogy pontosan melyik változó változása ébresztette fel őt.

Az `fd_global/3` meghívásakor és minden ébredéskor a rendszer elvégzi a felhasználó által megadott szűkítéseket. Ezeket a szűkítéseket a `clpfd:dispatch_global/4` többállományos (`multifile`) kampó-eljárás kibővítésével lehet megadni.

`clpfd:dispatch_global(Constraint, State, NewState, Actions)`
Ennek az eljárásnak a törzse definiálja a `Constraint` korlát ébredésekor végrehajtandó teendőket és állapot-változásokat. A `Constraint` paraméterben ugyanaz a struktúra fog megjelenni, mint amit az `fd_global/3` első paraméterében átadtunk. `State` tartalmazza az ébredéskor fennálló állapotot, `NewState`-et pedig nekünk kell majd kitölteni az új állapottal. A végrehajtandó szűkítéseket *tilos* a kampó-eljárás belsejében végrehajtani, helyette ezeket az `Actions` listában kell átadnunk, és ott kell jeleznünk a korlát sikeres lefutását vagy meghiúsulását is. Alapértelmezésben a korlát démona az eljárás lefutása után visszaalszik.

Az `Actions` lista az alábbi elemekből állhat (a sorrend nem számít):

- `exit` ill. `fail` — a korlát sikeresen ill. sikertelenül lefutott
- `X=V`, `X in R`, `X in_set S` — az adott szűkítést kérjük végrehajtani (ez is okozhat meghiúsulást)
- `call(Module:Goal)` — az adott hívást kérjük végrehajtani. A `Module:` modul-kvalifikáció kötelező!

Mivel a `dispatch_global/4` eljárás a többi `multifile` eljáráshoz hasonlóan interpretáltan fut, ezért a futás gyorsítása érdekében célszerű a `dispatch_global` eljárások törzsébe csak egyetlen klózt írni, ami az általunk írt korlátkezelő eljárásra mutat (mivel az már betöltéskor le fog fordulni, és így gyorsabb lesz a futás).

Az alábbi példa az `X #=< Y` korlát megvalósítása globális korlátként:

```
:- multifile clpfd:dispatch_global/4.
:- discontiguous clpfd:dispatch_global/4.   % nem folytonos eljárás

% X #=< Y, globális korlátként megvalósítva.
lseq(X, Y) :-
        % lseq(X,Y) globális démon indul, kezdőállapot: void.
        % Ébredés: X alsó és Y felső határának változásakor.
        fd_global(lseq(X,Y), void, [min(X),max(Y)]).

clpfd:dispatch_global(lseq(X,Y), St, St, Actions) :-
        dispatch_lseq(X, Y, Actions).

dispatch_lseq(X, Y, Actions) :-
        fd_min(X, MinX), fd_max(X, MaxX),
        fd_min(Y, MinY), fd_max(Y, MaxY),
```

```
        (   number(MaxX), number(MinY), MaxX =< MinY
            % buzgóbb, mint X#=<Y, mert az csak X vagy Y
            % behelyettesítésekor fut le.
        ->  Actions = [exit]
        ;   Actions = [X in inf..MaxY,Y in MinX..sup]
        ).
```

A fenti korlát működése igen egyszerű. Először meghatározzuk `X` és `Y` tartományainak szélső határait a megfelelő változókba. Ezek után ha `MaxX` és `MinY` is szám (tehát nem `inf` vagy `sup`), valamint `MaxX` kisebb vagy egyenlő, mint `MinY`, akkor befejezzük a működésünket, ellenkező esetben `X`-et az `inf..MaxY`, `Y`-t a `MinX..sup` intervallumra szűkítjük, és újra elalszunk. Ha az előző két szűkítés valamelyike meghiúsulna, akkor a Prolog automatikusan gondoskodik arról, hogy visszalépés következzen be.

Újabb példa, ezúttal az `S = sign(X)` (`X` előjele `S`) korlátra:

```
% X előjele S, globális korlátként megvalósítva.
sign(X, S) :-
        S in -1..1,
        fd_global(sign(X,S), void, [minmax(X),minmax(S)]).
        % Ébredés: X és S alsó és felső határának változásakor.

clpfd:dispatch_global(sign(X,S), St, St, Actions) :-
        fd_min(X, MinX0), sign_of(MinX0, MinS),
        fd_max(X, MaxX0), sign_of(MaxX0, MaxS),
        fd_min(S, MinS0), sign_min_max(MinS0, MinX, _),
        fd_max(S, MaxS0), sign_min_max(MaxS0, _, MaxX),
        Actions = [X in MinX..MaxX, S in MinS..MaxS|Exit],
        (   max(MinS0,MinS)=:=min(MaxS0,MaxS) -> Exit = [exit]
        ;   Exit = []
        ).

% sign_of(X, S): X egész vagy végtelen érték előjele S
sign_of(inf, S) :- !, S = -1.
sign_of(sup, S) :- !, S = 1.
sign_of(X, S) :- S is sign(X).

% sign_min_max(S, Min, Max): sign(x) = S  ⇔  x  ∈  Min..Max
sign_min_max(-1, inf, -1).
sign_min_max(0, 0, 0).
sign_min_max(1, 1, sup).
```

A reifikáció megvalósítása globális korláttal:

```
% X #=< Y #<=> B, globális korlátként megvalósítva.
lseq_reif(X, Y, B) :-
        B in 0..1, fd_global(lseq_reif(X,Y,B), void,
                             [minmax(X),minmax(Y),val(B)]).

clpfd:dispatch_global(lseq_reif(X,Y, B), St, St, Actions) :-
```

```
fd_min(X, MinX), fd_max(X, MaxX),
fd_min(Y, MinY), fd_max(Y, MaxY),
(   fdset_interval(_, MaxX, MinY)    % MaxX =< MinY
->  Actions = [exit,B=1]
;   empty_interval(MinX, MaxY)       % MaxY < MinX
->  Actions = [exit,B=0]
;   B == 1 -> Actions = [exit, call(user:lseq(X,Y))]
;   B == 0 -> Actions = [exit, call(user:less(Y,X))]
;   Actions = []
).
```

Ehhez hasonló trükkökkel természetesen tetszőleges globális korlátot átírhatunk olyan alakba, amely egy 0-1 értékű változóban tükrözi az igazságértékét, de ez nem „tiszta” reifikáció. Mindössze annyi ilyenkor a teendőnk, hogy meghatározzuk azokat a feltételeket, amelyekből kiderül, hogy a korlát, illetve a negáltja levezethető, és ezen feltételek teljesülése esetén az igazságértéket `0`-ra, illetve `1`-re kell szűkítenünk. Ugyanakkor arra is figyelni kell, hogy ha az igazságérték kerül behelyettesítésre, akkor a korlátot, illetve a negáltját ezúttal reifikáció nélkül kell felvennünk a tárba.

Valósítsuk meg globális korlátként a mágikus sorozatok példájában már használt `pontosan/3` korlátot! (Emlékeztetőül: `pontosan(I, L, E)` $\Leftrightarrow$ az `I` elem `L`-ben `E`-szer fordul elő)

```
% Az Xs listában az I szám pontosan N-szer fordul elő.
% N és az Xs lista elemei FD változók vagy számok lehetnek.
exactly(I, Xs, N) :-
    dom_susps(Xs, Susp),
    length(Xs, Len), N in 0..Len,
    fd_global(exactly(I,Xs,N), Xs/0, [minmax(N)|Susp]).
    % Állapot: L/Min ahol L az Xs-ből az I-vel azonos ill.
    % biztosan nem-egyenlő elemek esetleges kiszűrésével áll
    % elő, és Min a kiszűrt I-k száma.

% dom_susps(Xs, Susp): Susp dom(X)-ek listája, minden X ∈ Xs-re.
dom_susps([], []).
dom_susps([X|Xs], [dom(X)|Susp]) :-
    dom_susps(Xs, Susp).

clpfd:dispatch_global(exactly(I,_,N), Xs0/Min0, Xs/Min, Actions) :-
    ex_filter(Xs0, Xs, Min0, Min, I),
    length(Xs, Len), Max is Min+Len,
    fd_min(N, MinN), fd_max(N, MaxN),
    (   MaxN =:= Min -> Actions = [exit,N=MaxN|Ps],
        ex_neq(Xs, I, Ps)            % Ps = {x in_set \{I} | x ∈ Xs}
    ;   MinN =:= Max -> Actions = [exit,N=MinN|Ps],
        ex_eq(Xs, I, Ps)             % Ps = {x in_set  {I} | x ∈ Xs}
    ;   Actions = [N in Min..Max]
    ).

% ex_filter(Xs, Ys, N0, N, I): Xs-ből az I-vel azonos ill. attól
```

```
% biztosan különböző elemek elhagyásával kapjuk Ys-t,
% N-N0 a kiszűrt I-k száma.
ex_filter([], [], N, N, _).
ex_filter([X|Xs], Ys, N0, N, I) :-
    X==I, !, N1 is N0+1, ex_filter(Xs, Ys, N1, N, I).
ex_filter([X|Xs], Ys0, N0, N, I) :-
    fd_set(X, Set), fdset_member(I, Set), !,   % X még lehet I
    Ys0 = [X|Ys], ex_filter(Xs, Ys, N0, N, I).
ex_filter([_X|Xs], Ys, N0, N, I) :-           % X már nem lehet I
    ex_filter(Xs, Ys, N0, N, I).

% A Ps lista elemei `X in_set S', ∀ X ∈ Xs-re, S az \{I} FD halmaz.
ex_neq(Xs, I, Ps) :-
    fdset_singleton(Set0, I), fdset_complement(Set0, Set),
    eq_all(Xs, Set, Ps).

% A Ps lista elemei `X in_set S', ∀ X ∈ Xs-re, S az {I} FD halmaz.
ex_eq(Xs, I, Ps) :-
    fdset_singleton(Set, I), eq_all(Xs, Set, Ps).

% eq_all(Xs, S, Ps): Ps `X in_set S'-ek listája, minden X ∈ Xs-re.
eq_all([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :-
    eq_all(Xs, Set, Ps).
```

```
| ?- exactly(5, [A,B,C], N), N #=< 1, A=5.
A = 5, B in (inf..4)\/(6..sup), C in (inf..4)\/(6..sup), N = 1 ?
| ?- exactly(5, [A,B,C], N), A in 1..2, B in 3..4, N #>= 1.
A in 1..2, B in 3..4, C = 5, N = 1 ?
| ?- _L=[A,B,C], domain(_L,1,3), A #=< B, B #< C, exactly(3, _L, N).
A in 1..2, B in 1..2, C in 2..3, N in 0..1 ?
```

A SICStus 3.8.6-nál és a régebbi verzióknál a fenti megvalósítás kapcsán egy érdekes hibával találhatjuk magunkat szemközt:

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

Amint látható, a kapott megoldás hibás, hiszen a `[0,1]` listában a `0` elem nem 0-szor fordul elő, tehát az `exactly(0, L, N)` korlát nem áll fenn. A probléma általánosan a következőképpen fogalmazható meg:

> Legyen `c(X,Y)` egy globális korlát, amely `[dom(X),dom(Y)]` ébresztésű. Tegyük fel, hogy `X` tartománya változik, és ennek hatására a korlát szűkíti `Y` tartományát. Kérdés: ébredjen-e fel ettől újra a korlát?

A SICStus fejlesztői úgy döntöttek, hogy ilyen esetben a korlát ne ébredjen fel újra. Emiatt egy globális korláttal szemben támasztanunk kell egy olyan elvárást, hogy az *idempotens* legyen: ha meghívjuk, elvégezzük az akció-lista feldolgozását, majd azonnal újra meghívjuk, akkor a második hívás már biztosan ne váltson ki további szűkítéseket (tehát ne legyen érdemes újra meghívni). Formálisan: $dg(dg(s)) = dg(s)$, ahol dg a `dispatch_global/4` eljárásnak a tárra gyakorolt hatását jelöli.

Jelen példánkban a korlátunk megvalósítása nem teljesíti az idempotencia feltételét, mivel az `L` lista első eleme `N`, és ezáltal `N`-en keresztül az `exactly/3` második és harmadik paramétere „össze van kapcsolva". A SICStus a 3.8.7. verzió óta figyeli az összekapcsolt változókat, és ha ilyet talál, akkor automatikusan feltételezi a dg függvényről, hogy az nem idempotens, ezért újra és újra meghívja az `exactly/3` korlát démonát egészen addig, amíg van szűkítés. Így a második meghívás alkalmával már kiderül a fent megtalált megoldásról, hogy az hibás.

5.19.2. FD predikátumok

Az FD predikátumok segítségével egy korlát levezethetőségi és szűkítési szabályait írhatjuk le egy halmazértékű funkcionális nyelv alkalmazásával. Egy FD predikátum formailag hasonlít egy hagyományos Prolog predikátumhoz, de más a jelentése, és szigorúbb formai szabályokkal is szembe kell néznünk.

Az FD predikátumok mindig 1..4 klózból állnak, és mindnek más a „nyakjele". A `+:` nyakjelű klózt kötelező megírni, a `-:`, `+?` és `-?` nyakjelűek opcionálisak, akkor van rájuk szükség, ha reifikálható korlátot szeretnénk írni. A klózok törzse úgynevezett *indexikális*ok gyűjteményéből áll, az egyes indexikálisokat vesszővel kell egymástól elválasztani, de ez esetben a vessző *nem* konjunkciót jelent, a hagyományos Prolog predikátumokkal ellentétben. A `+:` és `-:` nyakjelű klózok *szűkítő* (*mondó*, *tell*) indexikálisokból állnak, és azt írják le, hogy az adott korlát, illetve a negáltja hogyan szűkíti a korlát-tárat. A `+?` és `-?` nyakjelű klózok egyetlen *kérdező* (*ask*) indexikálist tartalmaznak, amely azt írja le, hogy a korlát, illetve a negáltja mely feltétel teljesülése esetén vezethető le a tárból. Az FD klózok fejében az argumentumok kötelezően csak változók lehetnek, és a törzsben is csak ezek a változók szerepelhetnek. Példaként tekintsük az `X #=< Y` korlát FD predikátum változatát:

```
'x=<y'(X,Y) +:                  % Az X =< Y korlát szűkítései.
        X in inf..max(Y),       % X szűkítendő az inf..max(Y),
        Y in min(X)..sup.       % Y a min(X)..sup intervallumra.

'x=<y'(X,Y) -:                  % Az X =< Y korlát negáltjának,
        X in (min(Y)+1)..sup,   % azaz az X > Y korlátnak a
        Y in inf..(max(X)-1).   % szűkítései.

'x=<y'(X,Y) +?                  % Ha X tartománya része az
        X in inf..min(Y).       % inf..min(Y) intervallumnak,
                                % akkor X =< Y levezethető.

'x=<y'(X,Y) -?                  % Ha X tartománya része a
        X in (max(Y)+1)..sup.   % (max(Y)+1)..sup intervallumnak,
                                % akkor X > Y levezethető.
```

A fenti példából már láthatjuk, hogy az összes indexikális ***`Változó`*** `in` ***`TKif`*** alakú, ahol a ***`TKif`*** tartománykifejezés tartalmazza a ***`Változó`***-tól különböző *összes* fejváltozót. Ha olyan indexikálist írunk, amelyre ez utóbbi feltétel nem teljesül, akkor igen nagy a valószínűsége, hogy az indexikális hibásan fog működni (erre a 90. oldalon látunk is majd egy példát). A ***`TKif`*** tartománykifejezés (*range*) egy (parciális) halmazfüggvényt ír le, azaz a benne szereplő változók tartományának függvényében egy újabb tartományt állít elő. Például a `min(X)..sup` kifejezés az `X` alsó határának függvényében állít elő egy tartományt, ha `X`-ről azt tudjuk, hogy az `1..10` intervallumban van benne, akkor `min(X)..sup` = `1..sup` fog teljesülni. A ***`Változó`*** `in` *`TKif`* alakú kifejezés ***`Változó`*** értékét a *`TKif`* tartománykifejezés által előállított halmazra szűkíti (bizonyos feltételek fennállása esetén, ld. később).

Formálisan: az `X in` *`R`*`(Y,Z,…)` indexikális jelentése a következő reláció:

$$Rel(R) = \{\langle x, y, z, \ldots\rangle \,|x \in \mathtt{R}(\{y\}, \{z\}, \ldots)\}$$

Más szóval ha az *`R`*-beli változóknak egyelemű a tartománya, akkor az *`R`* tartománykifejezés értéke *pontosan* az adott relációt kielégítő `X` értékek halmaza lesz (ld. még a pont-szűkítés definícióját, 59. oldal).

Az FD predikátumok alapszabálya: az egy FD-klózon belül lévő indexikálisok jelentésének (azaz az általuk definiált relációnak) azonosnak kell lennie! Ennek oka az úgynevezett „*társasház-elv*": az FD predikátum kiértékelésére a Prolog *bármelyik* indexikálist felhasználhatja! Gyakorlásképp nézzük meg, hogy az előző példa FD-klózaiban teljesül-e ez az alapszabály:

```
'x=<y'(X, Y) +:
    X in inf..max(Y), % {⟨x,y⟩ |x ∈ inf..max({y})} ≡ {⟨x,y⟩ |x ∈ (−∞,y]} ≡ {⟨x,y⟩ |x ≤ y}
    Y in min(X)..sup. % {⟨x,y⟩ |y ∈ min({x})..sup} ≡ {⟨x,y⟩ |y ∈ [x,+∞)} ≡ {⟨x,y⟩ |y ≥ x}
```

Mivel $x \leq y$ és $y \geq x$ ekvivalensek, ezért itt a társasház-elv teljesül.

Most definiálni fogjuk a tartománykifejezések pontos szintaktikáját. Bevezetjük az alábbi jelöléseket (s továbbra is egy adott korlát-tárat fog jelenteni):

X egy korlát-változó, tartománya $D(X, s)$.

T egy számkifejezés (*term*), amelynek jelentése egy egész szám vagy egy végtelen érték, ezt $V(T, s)$-sel jelöljük. (Végtelen érték csak $T_1 \,..\, T_2$-ben lehet.)

R egy tartománykifejezés (*range*), amelynek jelentése egy számhalmaz, amit $S(R, s)$-sel jelölünk.

A tartománykifejezéseket alkotó elemi kifejezések és operátorok összefoglalva az alábbi táblázatban láthatóak:

Szintaxis	**Szemantika**
$T \Longrightarrow$	$V(T,s)$ =
integer	*integer* értéke
$\mid$ `inf`	$-\infty$
$\mid$ `sup`	$+\infty$
$\mid$ X	x feltéve, hogy $D(X,s)=\{x\}$. Egyébként az indexikális felfüggesztődik („pucér” változó esete).
$\mid$ $\mathtt{card}(X)$	$\lvert D(X,s)\rvert$ (a tartomány elemszáma)
$\mid$ $\mathtt{min}(X)$	$\min(D(X,s))$ (a tartomány alsó határa)
$\mid$ $\mathtt{max}(X)$	$\max(D(X,s))$ (a tartomány felső határa)
$\mid$ T_1 `+` T_2	$V(T_1,s)+V(T_2,s)$
$\mid$ T_1 `-` T_2	$V(T_1,s)-V(T_2,s)$
$\mid$ T_1 `*` T_2	$V(T_1,s)*V(T_2,s)$
$\mid$ T_1 `mod` T_2	$V(T_1,s) \bmod V(T_2,s)$
$\mid$ `-` T_1	$-V(T_1,s)$
$\mid$ T_1 `/>` T_2	$\lceil V(T_1,s)/V(T_2,s)\rceil$ (felfelé kerekített osztás)
$\mid$ T_1 `/<` T_2	$\lfloor V(T_1,s)/V(T_2,s)\rfloor$ (lefelé kerekített osztás)
$R \Longrightarrow$	$S(R,s)$ =
$\{T_1,\ldots,T_n\}$	$\{V(T_1,s),\ldots,V(T_n,s)\}$
$\mid$ $\mathtt{dom}(X)$	$D(X,s)$
$\mid$ T_1`..`T_2	$[V(T_1,s),V(T_2,s)]$ (intervallum)
$\mid$ R_1`/\`R_2	$S(R_1,s)\cap S(R_2,s)$ (metszet)
$\mid$ R_1`\/`R_2	$S(R_1,s)\cup S(R_2,s)$ (unió)
$\mid$ `\`R_1	$\setminus S(R_1,s)$ (komplementer halmaz)
$\mid$ `-` R_1	$\{-x \mid x\in S(R_1,s)\}$ (pontonkénti negáció)
$\mid$ R_1 `+` R_2	$\{x+y \mid x\in S(R_1,s), y\in S(R_2,s)\}$ (pont. összeg)
$\mid$ R_1 `+` T_2	$\{x+t \mid x\in S(R_1,s), t=V(T_2,s)\}$
$\mid$ R_1 `-` R_2	$\{x-y \mid x\in S(R_1,s), y\in S(R_2,s)\}$ (p. különbség)
$\mid$ R_1 `-` T_2	$\{x-t \mid x\in S(R_1,s), t=V(T_2,s)\}$
$\mid$ T_1 `-` R_2	$\{t-y \mid t=V(T_1,s), y\in S(R_2,s)\}$
$\mid$ R_1 `mod` R_2	$\{x \bmod y \mid x\in S(R_1,s), y\in S(R_2,s)\}$ (p. modulo)
$\mid$ R_1 `mod` T_2	$\{x \bmod t \mid x\in S(R_1,s), t=V(T_2,s)\}$
$\mid$ $\mathtt{unionof}(X, R_1, R_2)$	unió-kifejezés, ld. 91. oldal
$\mid$ $\mathtt{switch}(T, \mathit{MapList})$	kapcsoló-kifejezés, ld. 91. oldal
$\mid$ R_1 `?` R_2	feltételes kifejezés, ld. 92. oldal

Az ilyen kifejezésekben szereplő összeadás, kivonás, szorzás, osztás, modulus és ellentett műveletek mindegyike *pontonkénti* műveletvégzés. Ez azt takarja, hogy a műveletet végrehajtjuk a két operandusból a Descartes-szorzat segítségével kapott párokra, majd az eredményekből egy újabb halmazt képezünk. Például vegyük az alábbi korlátot:

```
f(X,Y) +: Y in 5 - dom(X).   % { 5-x | x ∈ dom(X) }
```

A fenti korlát az `Y #= 5-X` relációt valósítja meg, tartományszűkítő módon:

```
| ?- X in {1, 3, 5}, f(X,Y).
Y in{0}\/{2}\/{4} ?
```

Itt a korlát belsejében az `Y in 5 - dom(X)` hívás során minden $x \in D(X,s)$-re végrehajtódik az $y = 5 - x$ kivonás, majd ezeket az y-okat egy halmazba rakva kapjuk $D(Y,s)$-t.

A korábban `plusz/3` néven hivatkozott tartományszűkítő összegkorlát FD predikátummal való megvalósítása:

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

| ?- _X in {10,20}, _Y in {0,5}, _X+_Y #= Z.
Z in 10..25 ?
| ?- _X in {10,20}, _Y in {0,5}, 'x+y=t tsz'(_X, _Y, Z).
Z in{10}\/{15}\/{20}\/{25} ?
```

Példa „pucér” (indexikálisban önmagában álló) változóra:

```
f(X,Y,I) +: Y in \{X,X+I,X-I}.
% hasonló az N királynő feladat no_threat/3 korlátjához, ld. 47. oldal

| ?- X in {3, 5}, Y in 1..5, f(X, Y, 2), X = 3.
Y in {2}\{4} ?
```

Pucér változó használata esetén az indexikális végrehajtása felfüggesztődik addig, amíg a pucér változók be nem helyettesítődnek. Végül egy példa bonyolultabb számkifejezés indexikálisos megvalósítására:

```
| 'ax+c=t'(A,X,C,T) +:  % feltétel: A > 0
        X in (min(T) - C) /> A .. (max(T) - C) /< A,
        T in min(X)*A + C      ..  max(X)*A + C.

| ?- 'ax+c=t'(2,X,1,T), T in 0..4.
X in 0..1, T in 1..3 ?
```

5.19.3. Indexikálisok monotonitása

Az imént említettük azt is, hogy a ***Változó*** `in` ***TKif*** alakú indexikális ***Változó*** értékét csak bizonyos feltételek teljesülése mellett szűkíti a ***TKif*** tartománykifejezés által előállított halmazra. Most tisztázni fogjuk, hogy mik is ezek a bizonyos feltételek. Tekintsük az alábbi két FD predikátumot:

```
f(X, Y) +: Y in min(X)..sup.

| ?- X in 5..10, f(X, Y).
X in 5..10, Y in 5..sup?

f(X, Y) +: Y in max(X)..sup.

| ?- X in 5..10, f(X, Y).
X in 5..10, Y in inf..sup?
```

A két FD predikátum ránézésre nagyjából megegyezik, ha `X` tartománya egyelemű lenne, akkor mindkettő az `Y #>= X` korláttal ekvivalens jelentésű lenne. A második esetben azonban a Prolog mégsem hajlandó szűkíteni, ugyanis az `Y in 10..sup` szűkítést kéne végrehajtania, majd `X` tartományának későbbi szűkülésekor `Y` tartományát *bővítenie* kellene, ami nem lehetséges. Például ha a későbbiekben kiderülne, hogy `X in 6..7`, akkor `Y`-nak a `7..sup` tartományra kéne bővülnie.

Az általános megfogalmazáshoz vezessünk be néhány újabb fogalmat:

5.19.1. definíció: egy R tartománykifejezés egy s tárban *kiértékelhető*, ha az R-ben előforduló összes „pucér” változó tartománya az s tárban egyelemű (be van helyettesítve). A továbbiakban csak kiértékelhető tartománykifejezésekkel foglalkozunk.

5.19.2. definíció: egy s tárnak *pontosítása* s' ($s' \subseteq s$), ha minden X változóra $D(X, s') \subseteq D(X, s)$ (azaz s' szűkítéssel állhat elő s-ből).

5.19.3. definíció: egy R tartománykifejezés egy s tárra nézve *monoton*, ha minden $s' \subseteq s$ esetén $S(R, s') \subseteq S(R, s)$, azaz a tár szűkítésekor a kifejezés értéke is szűkül.

5.19.4. definíció: egy R tartománykifejezés egy s tárra nézve *antimonoton*, ha minden $s' \subseteq s$ esetén $S(R, s') \supseteq S(R, s)$.

5.19.5. definíció: R s-ben konstans, ha monoton és antimonoton (azaz s szűkülésekor már nem változik).

5.19.6. definíció: egy indexikálist monotonnak, antimonotonnak, ill. konstansnak nevezünk, ha a tartománykifejezése monoton, antimonoton, ill. konstans.

Példák

- `min(X)..max(Y)` egy tetszőleges tárban monoton.
- `max(X)..max(Y)` monoton minden olyan tárban, ahol `X` behelyettesített, és antimonoton, ahol `Y` behelyettesített.
- `card(X)..Y` kiértékelhető, ha `Y` behelyettesített, és ilyenkor antimonoton.
- `(min(X)..sup) \/ (0..sup)` egy tetszőleges tárban monoton, és konstans minden olyan tárban, ahol `min(X) >= 0`.

5.19.1. tétel: ha egy „X `in` R” indexikális monoton egy s tárban, akkor X értéktartománya az $S(R, s)$ tartománnyal szűkíthető.

Bizonyítás (vázlat): Tegyük fel, hogy $x_0 \in D(X, s)$ egy tetszőleges olyan érték, amelyhez találhatók olyan $y_0 \in D(Y, s)$, $z_0 \in D(Z, s)$, …értékek, hogy $\langle x_0, y_0, z_0, \ldots\rangle$ kielégíti az indexikális által definiált relációt. Azaz

$$\langle x_0, y_0, z_0, \ldots\rangle \in Rel(R) \Leftrightarrow x_0 \in S(R, s'), s' = \{Y \texttt{ in } \{y_0\}, Z \texttt{ in } \{z_0\}, \ldots\}$$

Itt $s' \subseteq s$, hiszen $y_0 \in D(Y, s)$, $z_0 \in D(Z, s)$, ... A monotonitás miatt $S(R, s) \supseteq S(R, s') \ni x_0$. Így tehát $S(R, s)$ tartalmazza az összes, a reláció által az s tárban megengedett értéket, ezért ezzel a halmazzal való szűkítés jogos.

A `clpfd` rendszer egy indexikálisról a következő irányelvek alapján dönti el, hogy az monoton-e vagy sem:

- Egy számkifejezésről egyszerűen megállapítható, hogy a tár szűkülésekor nő, csökken, vagy konstans-e (kivéve T_1 `mod` T_2, itt várunk, míg T_2 konstans lesz).
- Tartománykifejezések esetén:
  - T_1 `..` T_2 monoton, ha T_1 csökken és T_2 nő, antimonoton, ha T_1 nő és T_2 csökken.
  - `dom(`X`)` mindig monoton.
  - A metszet és unió műveletek eredménye (anti)monoton, ha mindkét operandusuk az, a komplemensképzés művelete megfordítja a monotonitást.
  - A pontonként végzett műveletek megőrzik az (anti)monotonitást (ehhez a T_i operandus konstans kell legyen, pl. `dom(X)+card(Y)`$\rightsquigarrow$`dom(X)+1`).
- Az (anti)monotonitás eldöntésekor a rendszer csak a változók behelyettesítettségét vizsgálja, pl. a `(min(X)..sup) \/ (0..sup)` kifejezést csak akkor tekinti konstansnak, ha `X` behelyettesített.

5.19.4. Szűkítő indexikálisok feldolgozási lépései

Egy `X in R` szűkítő indexikális feldolgozása mindig a végrehajthatóság vizsgálatával kezdődik: ha `R`-ben behelyettesítetlen („pucér”) változó van, vagy `R`-ről a rendszer nem látja azonnal, hogy monoton, akkor felfüggeszti a végrehajtását addig, amíg ezek a feltételek nem teljesülnek. Ezek után meghatározza az indexikálisból képződő démon aktiválási feltételeit az egyes `R`-beli változókra nézve, mégpedig az alábbiak szerint (`Y` az `R`-ben előforduló változók egyike):

- `dom(Y), card(Y)` környezetben előforduló `Y` változó esetén az indexikális a változó tartományának bármilyen módosulásakor aktiválandó;
- `min(Y)` környezetben – alsó határ változásakor aktiválandó;
- `max(Y)` környezetben– felső határ változásakor aktiválandó.

A szűkítés menete a következőek szerint történik: ha $D(X, s)$ és $S(R, s)$ diszjunktak, akkor visszalépés következik be, egyébként a tárat az X `in` $S(R, s)$ korláttal szűkítjük (erősítjük), azaz $D(X, s) := D(X, s) \cap S(R, s)$. A démon akkor fejezi be működését, ha az `R` tartománykifejezés konstanssá válik (például azért, mert minden `R`-beli változó behelyettesítődik). Ekkor Rel(`R`) garantáltan fennáll, ezért *az indexikálist tartalmazó korlát* levezethető, ilyenkor viszont a társasház-elv alapján hatékonysági okokból a korlát *összes* indexikálisa befejezi a működését.

Az indexikálisok feldolgozási lépéseit néhány példán keresztül is bemutatjuk:

```
'x=<y'(X, Y) +:
        X in inf..max(Y),      % (ind1)
        Y in min(X)..sup.      % (ind2)
```

Az *(ind1)* indexikális végrehajtási lépései

- Végrehajthatóság vizsgálata: nincs benne pucér változó, monoton, tehát végrehajtható
- Aktiválás: `Y` felső határának változásakor.
- Szűkítés: `X` tartományát elmetsszük az `inf..max(Y)` tartománnyal, azaz `X` felső határát az `Y`-éra állítjuk, ha az utóbbi a kisebb.
- Befejezés: amikor `Y` behelyettesítődik, akkor *(ind1)* konstanssá válik. Ekkor **mindkét** indexikális — *(ind1)* és *(ind2)* is —befejezi működését.

Egy másik korlát, kicsit kevésbé részletesen:

```
'abs(x-y)>=c'(X, Y, C) +:
        X in (inf .. max(Y)-C) \/ (min(Y)+C .. sup),
        % vagy:  X in \ (max(Y)-C+1 .. min(Y)+C-1),
        Y in (inf .. max(X)-C) \/ (min(X)+C .. sup).

| ?- 'abs(x-y)>=c'(X,Y,5), X in 0..6.
X in 0..6, Y in(inf..1)\/(5..sup) ?
| ?- 'abs(x-y)>=c'(X,Y,5), X in 0..9.
X in 0..9, Y in inf..sup ?
```

A `no_threat/3` korlát (ld. N királynő feladat, 47. oldal) kicsit erősebb indexikálisos megvalósítása:

```
no_threat_2(X, Y, I) +:
        X in \{Y,Y+I,Y-I}, Y in \{X,X+I,X-I}.

| ?- no_threat_2(X, Y, 2), Y in 1..5, X=3.
X = 3, Y in {2}\/{4} ?
| ?- no_threat_2(X, Y, 2), Y in 1..5, X in {3,5}.
X in{3}\/{5}, Y in 1..5 ?
```

Érdemes megfigyelni, hogy a második példában nincs szűkítés annak ellenére, hogy `Y` sem 3, sem 5 nem lehet. Azonban mivel az `Y`-hoz tartozó indexikálisban `X` pucéron szerepel, de még nem teljesen behelyettesített, ezért a teljes indexikális felfüggesztődik.

Végül nézzünk egy példát arra az esetre, amikor a társasház-elv nem érvényesül, és ezért az FD predikátum hibásan működik:

```
'x=<y=<z rossz'(X, Y, Z) +:
        Y in min(X)..max(Z),     % { ⟨x,y,z⟩ | x ≤ y ≤ z}
        Z in min(Y).. sup,       % { ⟨x,y,z⟩ |     y ≤ z}
        X in inf..max(Y).        % { ⟨x,y,z⟩ | x ≤ y    }

| ?- 'x=<y=<z rossz'(15, 5, Z).
Z in 5..sup ?
```

A korlát felvételekor egyedül a második indexikális tud aktiválódni (mivel `Y` és `X` már eleve konstans), és ez leszűkíti `Z`-t az `5..sup` intervallumra anélkül, hogy figyelembe venné, hogy a korlát a `15` $\not\leq$ `5` feltétel miatt eleve nem állhat fenn. A javításhoz meg kell ismerkednünk azzal a három tartománykifejezéssel is, amelyekről eddig még nem esett szó.

5.19.5. Bonyolultabb tartománykifejezések

Unió-kifejezés: `unionof(X, H, T)`

Egy `unionof(X, H, T)` kifejezésben `X` változó, `H` és `T` tartománykifejezések. Kiértékelése egy s tárban: legyen `H` értéke az s tárban $S(\mathtt{H}, s) = \{x_1, \ldots, x_n\}$ (ha $S(H, s)$ végtelen, a kiértékelést felfüggesztjük). Képezzük a T_i kifejezéseket úgy, hogy `T`-ben `X` helyébe x_i-t írjuk. Ekkor az unió-kifejezés értéke a $S(T_1, s), \ldots, S(T_n, s)$ halmazok uniója. Képlettel:

$$S(\mathtt{unionof(X,H,T)}, s) = \bigcup\{S(\mathtt{T}, (s \wedge \mathtt{X} = x))|x \in D(\mathtt{H}, s)\}$$

Egy unió-kifejezés kiértékelésének ideje/tárigénye arányos a `H` tartomány méretével!

A `no_threat/3` (ld. N királynő feladat, 47. oldal) maximálisan szűkítő, de egyáltalán nem hatékony megvalósítása:

```
no_threat_3(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}).

| ?- no_threat_3(X, Y, 2), Y in 1..5, X in {3,5}.
X in {3,5}, Y in {1,2,4} ?
```

Kapcsoló-kifejezés: `switch(T, MapList)`

`T` egy számkifejezés, `MapList` pedig *`integer`*-`Range` alakú párokból álló lista, ahol az *`integer`* értékek mind különböznek (`Range` egy tartománykifejezés). Jelöljük K-val $V(T, s)$-t (ha `T` nem kiértékelhető, az indexikálist felfüggesztjük). Ha `MapList` tartalmaz egy $K - R$ párt, akkor a kapcsoló-kifejezés értéke $S(R, s)$ lesz, egyébként az üres halmaz lesz az értéke. Példa:

```
% Ha I páros, Z = X, egyébként Z = Y. Vár míg I értéket nem kap.
p(I, X, Y, Z) +:   Z in switch(I mod 2, [0-dom(X),1-dom(Y)]).

p2(I, X, Y, Z) +:  % ugyanaz mint p/4, de nem vár.
    Z in unionof(J, dom(I) mod 2, switch(J, [0-dom(X),1-dom(Y)])).
```

Egy `relation/3` kapcsolat megvalósítható egy `unionof-switch` szerkezettel:

```
% relation(X, [0-{1},1-{0,2},2-{1,3},3-{2}], Y) ⇔  |x − y| = 1  x,y ∈ [0,3]
absdiff1(X, Y) +:
  X in unionof(B,dom(Y),switch(B,[0-{1},1-{0,2},2-{1,3},3-{2}])),
  Y in unionof(B,dom(X),switch(B,[0-{1},1-{0,2},2-{1,3},3-{2}])).
```

Példa: az `Y in {0,2,4}` tárban `absdiff1` első indexikálisának kiértékelése a következő (jelöljük `MAPL`-lel a `[0-{1},1-{0,2},2-{1,3},3-{2}]` listát):

```
X in unionof(B,{0,2,4},switch(B,MAPL)) =
     switch(0,MAPL) \/ switch(2,MAPL) \/ switch(4,MAPL) =
     {1}            \/ {1,3}          \/ {}             = {1,3}
```

Feltételes kifejezés: `Felt ? Tart`

`Felt` és `Tart` tartománykifejezések. Ha $S($`Felt`$, s)$ üres halmaz, akkor a feltételes kifejezés értéke is üres halmaz, egyébként pedig azonos $S($`Tart`$, s)$ értékével. Példák:

```
% X in 4..8 #<=> B.
'x in 4..8<=>b'(X, B) +:
        B in (dom(X)/\(4..8)) ? {1} \/ (dom(X)/\ \(4..8)) ? {0},
        X in (dom(B)/\{1}) ? (4..8) \/ (dom(B)/\{0}) ? \(4..8).
```

A feltételes kifejezés használatával már meg tudjuk fogalmazni az `'x=<y=<z'/3` korlátunk helyes szűkítési feltételeit:

```
'x=<y=<z'(X, Y, Z) +:
        Y in min(X)..max(Z),
        Z in ((inf..max(Y)) /\ dom(X)) ? (min(Y)..sup),  % (*)
            % ha max(Y) ≥ min(X) akkor min(Y)..sup egyébként {}
        X in ((min(Y)..sup) /\ dom(Z)) ? (inf..max(Y)).
```

A `(*)` indexikális jobboldalának kiértékelése az előzőleg problematikus esetben (`X = 15, Y = 5`):

```
X = 15, Y = 5 ⟹ (inf..5)/\{15} ? (5..sup) = {} ? (5..sup) = {}
X = 15, Y in 5..30 ⟹ (inf..30)/\{15} ? 5..sup = 15 ? 5..sup = 5..sup
```

A feltételes kifejezés a kiértékelés késleltetésére is használható, ha `(Felt?(inf..sup) \/ Tart)` alakban használjuk. Ezen tartománykifejezés értéke $S($`Tart`$, s)$, ha $S($`Felt`$, s)$ üres, egyébként `inf..sup`. Az ilyen szerkezetekben `Tart` értékét a rendszer nem értékeli ki, amíg `Felt` nem üres. Példa:

```
% Maximálisan szűkít, kicsit kevésbé lassú
no_threat_4(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}).
```

Ez a `no_threat/3` korlát egy olyan megvalósítása, amely csak abban az esetben használja az egyik változó esetében az `unionof` szerkezetet, ha a másik változó halmazának számossága már 4-nél kisebb. A `(**)` indexikális jobb oldalának kiértékelése (`I = 1`):

```
Y in 5..8 ---> (4..4)?(inf..sup) \/ unionof(...) = inf..sup

Y in 5..7 ---> (4..3)?(inf..sup) \/ unionof(B,5..7,\{B,B+1,B-1}) =
                {}?(inf..sup) \/ unionof(B,5..7,\{B,B+1,B-1}) =
                {} \/ \{5,6,4} \/ \{6,7,5} \/ \{7,8,6} = \{6}
```

5.19.6. Reifikálható FD predikátumok

Egy reifikálható FD predikátumban általában mind a négy nyakjelű klóz szerepel. Ha valamelyiket elhagyjuk, akkor az ahhoz tartozó szűkítés, illetve levezethetőség-vizsgálat elmarad. Emlékeztetőül: a `+:` és `-:` nyakjelű klózok *szűkítő* (*mondó*, *tell*) indexikálisokból állnak, és azt írják le, hogy az adott korlát, illetve a negáltja hogyan szűkíti a korlát-tárat. A `+?` és `-?` nyakjelű klózok egyetlen *kérdező* (*ask*) indexikálist tartalmaznak, amely azt írja le, hogy a korlát, illetve a negáltja mely feltétel teljesülése esetén vezethető le a tárból. A kérdező klózban egy `X in R` kérdező indexikális valójában a `dom(X)` $\subseteq$ `R` feltételt fejezi ki, mint az FD predikátum (vagy negáltja) levezethetőségi feltételét. Például az `X #\= Y` korlát esetén:

```
'x\\=y'(X,Y) +:         % 1. a korlátot szűkítő indexikálisok
        X in \{Y},
        Y in \{X}.

'x\\=y'(X,Y) -:         % 2. a negáltját szűkítő 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 (itt felesleges, az okot
                        % lásd később)
```

Egy `X #\= Y #<=> B` reifikáció ezek után a következőképpen megy végbe: a 3. klóz folyamatosan figyeli, hogy `X` és `Y` tartományai diszjunktak-e (`dom(X)` $\subseteq$ `\dom(Y)`), és ha ez teljesül, akkor `B`-be 1-et helyettesít. Ugyanakkor a 4. klóz figyeli, hogy `X=Y` igaz-e (`dom(X)` $\subseteq$ `{Y}`), és ha igen, akkor `B`-be 0-t helyettesít. Közben egy külön démon figyeli, hogy `B` behelyettesítődik-e, ha igen, és `B=1`, akkor elindítja az 1. klózbeli indexikálisokat. `B=0` esetben a 2. klóz indexikálisai indulnak el.

5.19.7. Kérdező indexikálisok feldolgozási lépései

A kérdező indexikálisokra másfajta feldolgozási szabályok érvényesek, mint a szűkítő indexikálisokra. A legfontosabb különbség, hogy egy kérdező indexikális végrehajtását mindaddig felfüggesztjük, amíg kiértékelhető és *antimonoton* nem lesz (ellentétben a szűkítő indexikálisokkal, ahol a monotonitás volt a feltétel). Az ébresztési feltételek a szűkítő indexikálisokhoz hasonlóak (`Y` az `X in R` kifejezés esetén egy `R`-ben előforduló változó):

- `X` tartományának bármilyen változásakor
- `dom(Y)`, `card(Y)` környezetben előforduló `Y` változó esetén az indexikális a változó tartományának bármilyen módosulásakor aktiválandó;
- `min(Y)` környezetben – alsó határ változásakor aktiválandó;
- `max(Y)` környezetben – felső határ változásakor aktiválandó.

Ha az indexikális felébred, két eset lehetséges:

- Ha $D(X,s) \subseteq S(R,s)$, akkor a korlát levezethetővé vált.
- Ha $D(X,s)$ és $S(R,s)$ diszjunktak, valamint $S(R,s)$ monoton is (vagyis konstans, mivel idáig csak akkor juthattunk el, ha antimonoton is), akkor a korlát negáltja levezethetővé vált (emiatt felesleges az `'x\\=y'` FD predikátum 4. klóza).
- Egyébként újra elaltatjuk az indexikálist.

Egy egyszerű példa:

```
'x=<y'(X,Y) +?
        X in inf..min(Y).      % (ind1)
```

Az *(ind1)* kérdező indexikális végrehajtási lépései

- Végrehajthatóság vizsgálata: nincs benne pucér változó, minden tárban antimonoton, tehát végrehajtható
- Aktiválás: `Y` alsó határának változásakor.
- Levezethetőség: megvizsgáljuk, hogy `X` tartománya része-e az `inf..min(Y)` tartománynak, azaz `max(X) =< min(Y)` fennáll-e. Ha igen, akkor a korlát levezethetővé vált, a démon befejezi működését, és a reifikációs változó az `1` értéket kapja.
- Negált levezethetősége: megvizsgáljuk, hogy a tartománykifejezés konstans-e, azaz `Y` behelyettesített-e. Ha igen, akkor megvizsgáljuk, hogy az `inf..min(Y)` intervallum és `X` tartománya diszjunktak-e, azaz `Y < min(X)` fennáll-e. Ha mindez teljesült, akkor a korlát negáltja levezethetővé vált, a démon befejezi működését, és a reifikációs változó a `0` értéket kapja.

5.19.8. Korlátok automatikus fordítása indexikálisokká

A SICStus lehetőséget kínál arra, hogy egy egyszerű `clpfd` korlátot automatikusan indexikálissá fordítsunk. A következő korlátozások érvényesek:

- Az indexikálissá fordítandó kifejezés formája `Head +: Korlát.`, ahol `Korlát` csak lineáris kifejezésekből álló *aritmetikai* korlát vagy a `relation/3` és `element/3` szimbolikus korlátok egyike lehet. A `relation/3` és `element/3` korlátok unió- és kapcsoló-kifejezésekké fordulnak (ld. 91. oldal). Mivel ezek végrehajtási ideje erősen függ a tartomány méretétől, ezért vigyázni kell a kezdő tartományok megfelelő beállítására.
- Csak a `+:` nyakjel használható, így ezek a korlátok nem reifikálhatóak.

Az így kapott átfordított korlátok a változók számának függvényében négyzetes helyigényűek (szemben az eredeti korlátok lineáris helyigényével) és általában lassabbak is. Előfordulhat azonban olyan eset is, hogy az átfordított változat gyorsabb, mint ahogy a később ismertetésre kerülő torpedó és dominó feladatok esetén is:

Torpedó	`:-`	`+:`
fules2	12.31	10.67
dense-clean	4.02	2.77
dense-collapse	1.79	1.29

Dominó	`:-`	`+:`
2803	174.7	127.6
2804	37.3	27.7
2805	327.7	239.8

A torpedó feladatban a `relation/3` korlátot, a dominó feladatban a `B1+...+BN #= 1` alakú korlátokat (`Bi 0..1` értékű változók, `N=<5`) fejtettünk ki indexikálisokká.

5.19.9. Indexikálisok összefoglalása

Legyen $\texttt{C(Y}_1\texttt{, ..., Y}_n\texttt{)}$ egy FD-predikátum, amelyben szerepel egy

$$\texttt{Y}_i\ \texttt{in R(Y}_1\texttt{, ..., Y}_{i-1}\texttt{, Y}_{i+1}\texttt{, ..., Y}_n\texttt{)}$$

indexikális. Az `R` tartománykifejezés által definiált reláció:

$$C = \{\langle y_1, \ldots, y_n\rangle \,|y_i \in S(\mathtt{R}, \langle \mathtt{Y}_1 = y_1, \ldots, \mathtt{Y}_{i-1} = y_{i-1}, \mathtt{Y}_{i+1} = y_{i+1}, \ldots\rangle)\}$$

Kiterjesztett alapszabály: Egy FD-predikátum csak akkor értelmes, ha a pozitív (`+:` és `+?` nyakjelű) klózaiban levő összes indexikális ugyanazt a relációt definiálja, továbbá a negatív (`-:` és `-?` nyakjelű) klózaiban levő összes indexikális ennek a relációnak a negáltját (komplemensét) definiálja.

Ha R monoton egy s tárra nézve, akkor $S(R, s)$-ről belátható, hogy minden olyan y_i értéket tartalmaz, amelyek (az s által megengedett y_j értékekkel együtt) a C relációt kielégítik. Ezért szűkítő indexikálisok esetén jogos az Y_i tartományát $S(R, s)$-rel szűkíteni. Ha viszont R antimonoton egy s tárra nézve, akkor $S(R, s)$-ről belátható, hogy minden olyan y_i értéket kizár, amelyekre (az s által megengedett legalább egy y_j érték-rendszerrel együtt) a C reláció nem áll fenn. Ezért kérdező indexikálisok esetén, ha $D(Y_i, s) \subseteq S(R, s)$, jogos a korlátot az s tárból levezethetőnek tekinteni. A fentiek miatt természetesen adódik az indexikálisok felfüggesztési szabálya: a szűkítő indexikálisok végrehajtását mindaddig felfüggesztjük, amíg monotonná nem válnak; a kérdező indexikálisok végrehajtását mindaddig felfüggesztjük, amíg 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 az 1 értékre tükröződni (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.

6. fejezet

Az `fdbg` nyomkövető csomag

Ebben a fejezetben a `clpfd` programok nyomkövetésére szolgáló `fdbg` csomagot mutatjuk be részletesebben. Az `fdbg` könyvtár megalkotásakor a szerzők (*Szeredi Tamás* és *Hanák Dávid*) az alábbi szempontokat tartották szem előtt:

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

6.1. Alapfogalmak

A csomag használatához szükséges megismerkednünk néhány alapvető fogalommal:

6.1.1. definíció: *`clpfd` esemény*nek nevezzük egy globális korlát felébredését vagy bármely címkézési eseményt (címkézés kezdete, címkézési lépés vagy címkézés meghiúsulása).

6.1.2. definíció: *Megjelenítő*nek (*visualizer*) nevezzük a `clpfd` eseményekre reagáló predikátumot. Általában az a feladata, hogy az eseményt valamilyen formában kiírja. A tényleges esemény bekövetkezése és így az esemény hatásainak érvényesülése *előtt* hívódik meg. Mindkét fajta `clpfd` eseményhez külön megjelenítő tartozik, így beszélhetünk *korlát-megjelenítő*ről és *címkézés-megjelenítő*ről.

6.1.3. definíció: *Jelmagyarázat*nak (*legend*) nevezzük az éppen megfigyelt korlát után kiíródó szövegrészt, amely rendszerint a korlátban részt vevő változókat, a hozzájuk kapcsolódó tartományokat és a vizsgált korlát végrehajtásával kapcsolatos következtetéseket tartalmazza.

Az `fdbg` által csak a globális korlátok követhetőek nyomon, az indexikálisok nem, viszont a globális korlátok közül a felhasználói és a beépített korlátok ugyanolyan módon kezelhetőek. Az `fdbg` nyomkövetés bekapcsolt állapota esetén a formula-korlátokból mindenképp globális korlátok képződnek, nem pedig indexikálisok.

Az `fdbg` könyvtár külön segédeljárásokat biztosít a kifejezésekben található FD változók megjelöléséhez (*annotálás*), az annotált kifejezések jól olvasható megjelenítéséhez, valamint jelmagyarázat előkészítéséhez és kiírásához.

Az `fdbg` könyvtár szolgáltatásait az alábbi paranccsal lehet igénybe venni:

```
| ?- use_module(library(fdbg)).
```

6.2. A nyomkövetés be- és kikapcsolása

- `fdbg_on`
  `fdbg_on(+Options)`
  Engedélyezi a nyomkövetést alapértelmezett vagy megadott beállításokkal. A nyomkövetést az `fdbg_output` álnevű (stream alias) folyamra írja a rendszer, alaphelyzetben ez a pillanatnyi kimeneti folyam (*current output stream*) lesz. Legfontosabb opciók:

  - `file(Filename, Mode)`
    A megjelenítők kimenete a ***Filename*** nevű állományba irányítódik át, amely az `fdbg_on/1` hívásakor nyílik meg ***Mode*** módban. `Mode` lehet `write` vagy `append`.
  - `stream(Stream)`
    A megjelenítők kimenete a ***Stream*** folyamra irányítódik át.
  - `constraint_hook(Goal)`
    Goal két argumentummal kiegészítve meghívódik a korlátok felébredésekor. Alapértelmezésben ez az `fdbg_show/2` eljárás, ld. később. Ezzel a paraméterrel lehet felüldefiniálni a korlát-megjelenítőt.
  - `labeling_hook(Goal)`
    Goal három argumentummal kiegészítve meghívódik minden címkézési eseménykor. Alapértelmezésben ez az `fdbg_label_show/3`, ld. később. Ezzel a paraméterrel lehet felüldefiniálni a címkézés-megjelenítőt.
  - `no_constraint_hook`, `no_labeling_hook`
    Nem lesz adott fajtájú megjelenítő.

- `fdbg_off`
  Kikapcsolja a nyomkövetést, lezárja a `file` opció hatására megnyitott állományt (ha volt ilyen).

Kimenet átirányítása, beépített megjelenítő, nincs címkézési nyomkövetés:

```
| ?- fdbg_on([file('my_log.txt', append), no_labeling_hook]).
```

Kimenet átirányítása szabványos folyamra, saját és beépített megjelenítő együttes használata:

```
| ?- fdbg_on([constraint_hook(fdbg_show), constraint_hook(my_show),
             stream(user_error)]).
```

6.3. Kifejezések elnevezése

Az `fdbg` által produkált kimeneten az FD változók alaphelyzetben <fdvar_1>, <fdvar_2>, …formában jelennek meg, ami megnehezíti az olvashatóságot. Éppen ezért minden FD változóhoz egy saját nevet rendelhetünk, ami a kimeneten az <fdvar_*x*> forma helyett fog megjelenni. Lehetőség van arra is, hogy egy egész kifejezéshez rendeljünk nevet. Egy kifejezés elnevezésekor a kifejezésben szereplő összes változóhoz egy-egy származtatott név rendelődik – ez a név a megadott névből és a változó kiválasztójából keletkezik (struktúra argumentum-sorszámok ill. lista indexek sorozata). A létrehozott nevek egy globális listába kerülnek, amely mindig egyetlen toplevel híváshoz tartozik, nem vivődik át a következő toplevel hívásra (*illékony*).

Predikátumok

- **`fdbg_assign_name(+Name, +Term)`**
  A *`Term`* kifejezéshez a *`Name`* nevet rendeli az aktuális toplevel hívásban.
- **`fdbg_current_name(?Name, -Term)`**
  Lekérdez egy kifejezést (változót) a globális listából a neve alapján. Ha *`Name`* változó, akkor felsorolja az összes tárolt név-kifejezés párt.
- **`fdbg_get_name(+Term, -Name)`**
  `Name` a *`Term`* kifejezéshez rendelt név. Ha *`Term`*-nek még nincs neve, automatikusan hozzárendelődik egy.

Pl. `fdbg_assign_name(foo, bar(A, [B, C]))` hatására a következő nevek generálódnak:

név	kifejezés	megjegyzés
`foo`	`bar(A, [B, C])`	a teljes kifejezés
`foo_1`	`A`	`bar` első argumentuma
`foo_2_1`	`B`	`bar` második argumentumának első eleme
`foo_2_2`	`C`	`bar` második argumentumának második eleme

6.4. Egyszerűbb `fdbg` nyomkövetési példák

```
| ?- use_module([library(clpfd),library(fdbg)]).

| ?- fdbg_on.
% The clp(fd) debugger is switched on
% advice
| ?- Xs=[X1,X2], fdbg_assign_name(Xs, 'X'),
     domain(Xs, 1, 6), X1+X2 #= 8, X2 #>= 2*X1+1.

domain([<X_1>,<X_2>],1,6)           X_1 = inf..sup -> 1..6
                                    X_2 = inf..sup -> 1..6
                                    Constraint exited.

<X_1>+<X_2>#=8                      X_1 = 1..6 -> 2..6
                                    X_2 = 1..6 -> 2..6

<X_2>#>=2*<X_1>+1                   X_2 = 2..6 -> 5..6
                                    X_1 = 2..6 -> {2}
                                    Constraint exited.

<X_2>#=6    [2+<X_2>#=8 (*)]        X_2 = 5..6 -> {6}
                                    Constraint exited.
```

```
X1 = 2, X2 = 6 ?
% advice
```

A (*) olvashatóbb alak a `library(fdbg)` négy sorának kikommentezésével állítható elő.

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

<fdvar_1> in 1..4                    fdvar_1 = inf..sup -> 1..4
                                     Constraint exited.

Labeling [2, <fdvar_1>]: starting in range 1..4.
Labeling [2, <fdvar_1>]: bisect: <fdvar_1> =< 2
        Labeling [4, <fdvar_1>]: starting in range 1..2.
        Labeling [4, <fdvar_1>]: bisect: <fdvar_1> =< 1
X = 1 ? ;
        Labeling [4, <fdvar_1>]: bisect: <fdvar_1> >= 2
X = 2 ? ;
        Labeling [4, <fdvar_1>]: failed.
Labeling [2, <fdvar_1>]: bisect: <fdvar_1> >= 3
        Labeling [8, <fdvar_1>]: starting in range 3..4.
        Labeling [8, <fdvar_1>]: bisect: <fdvar_1> =< 3
X = 3 ? ;
        Labeling [8, <fdvar_1>]: bisect: <fdvar_1> >= 4
X = 4 ? ;
        Labeling [8, <fdvar_1>]: failed.
Labeling [2, <fdvar_1>]: failed.
no
```

6.5. Beépített megjelenítők

Az `fdbg` könyvtár egy-egy alapértelmezett megjelenítőt bocsájt a felhasználó rendelkezésére. A korlát-megjelenítőt az `fdbg_show/2`, a címkézés-megjelenítőt az `fdbg_label_show/3` predikátum tartalmazza. Ha az `fdbg_on` predikátumnak nem adunk meg külön korlát-, illetve címkézés-megjelenítőt, akkor ezeket a megjelenítőket alkalmazza.

- **`fdbg_show(+`*`Constraint`*`, +`*`Actions`*`)`**
  Beépített korlát-megjelenítő. A `dispatch_global`-ból való kilépéskor hívódik meg. Megkapja az aktuális korlátot és az általa előállított akciólistát, majd ennek alapján megjeleníti a korlátot és a hozzá tartozó jelmagyarázatot.

  „Szimulált” példa-hívás `X1=3` és `X2=3` szűkítésekkel:

  ```
  | ?- Xs=[X1,X2,X3], fdbg_assign_name(Xs, 'X'),
       domain(Xs, 1, 3), X3 #\= 3, fdbg_on,
       fdbg_show(exactly(3,Xs,2),[exit,X1=3,X2=3]).

  exactly(3,[<X_1>,<X_2>,<X_3>],2)
      X_1 = 1..3 -> {3}
  ```

```
X_2 = 1..3 -> {3}
X_3 = 1..2
Constraint exited.
```

- **`fdbg_label_show(+Event, +ID, +Variable)`**
  Beépített címkézés-megjelenítő. Címkézési eseménykor (kezdet, szűkítés, meghiúsulás) hívódik meg. *`Event`*-ben megkapja a hívást kiváltó eseményt, *`ID`*-ben a címkézési lépés azonosítóját, *`Variable`*-ben pedig magát a címkézendő változót. *`Event`* megegyezik a `start` atommal, ha a kiváltó esemény a címkézés kezdete, megegyezik a `fail` atommal, ha a kiváltó esemény a címkézés meghiúsulása. Példa:

```
| ?- fdbg_assign_name(X, 'X'), X in {1,3}, fdbg_on,
     indomain(X).
% The clp(fd) debugger is switched on
Labeling [1, <X>]: starting in range {1}\/{3}.
Labeling [1, <X>]: indomain_up: <X> = 1

X = 1 ? ;
Labeling [1, <X>]: indomain_up: <X> = 3

X = 3 ? ;
Labeling [1, <X>]: failed.

no
```

A fenti kimenet elkészítése során végrehajtott megjelenítő-hívások:

```
fdbg_label_show(start,1,X)
fdbg_label_show(step('$labeling_step'(X,=,1,indomain_up)),1,X)
fdbg_label_show(step('$labeling_step'(X,=,3,indomain_up)),1,X)
fdbg_label_show(fail,1,X)
```

6.6. Testreszabás kampó-eljárásokkal

Az alábbi kampó-eljárások a következő három argumentummal rendelkeznek:

- *Name* — az FD változó neve
- *Variable* — maga a változó
- *FDSetAfter* — a változó tartománya, *miután* az aktuális korlát elvégezte rajta a szűkítéseket

A kampó-eljárások felülbírálásához azokat *multifile* eljárásként kell definiálni.

- **`fdbg:fdvar_portray(+Name, +Variable, +FDSetAfter)`**
  A kiírt korlátokban szereplő változók megjelenésének megváltoztatására szolgál. Az alapértelmezett viselkedés *Name* kiírása kacsacsőrök között. Az alábbi példa a változó neve mellett a változó régi tartományát is kiírja:

```
:- multifile fdbg:fdvar_portray/3.

fdbg:fdvar_portray(Name, Var, _) :-
       fd_set(Var, Set), fdset_to_range(Set, Range),
       format('<~p = ~p>', [Name,Range]).
```

- **fdbg:legend_portray(+*Name*, +*Variable*, +*FDSetAfter*)**
  A jelmagyarázat minden sorára meghívódik, és így végigfut a jelmagyarázatban szereplő változókon. A sorokat alapértelmezésben négy szóköz nyitja és egy újsor karakter zárja, ezeket nem kell kiírni, de nem is lehet őket felülbírálni. Az alábbi példa a változó tartományának szűkülését lista formában tünteti fel:

```
:- multifile fdbg:legend_portray/3.

fdbg:legend_portray(Name, Var, Set) :-
        fd_set(Var, Set0), fdset_to_list(Set0, L0),
        (   Set0 == Set
        ->  format("~p = ~p", [Name, L0])
        ;   fdset_to_list(Set, L),
            format("~p = ~p -> ~p", [Name,L0,L])
        ).
```

A példák kimenete összevetve az alapértelmezettel:

Eredeti alak	*Testreszabott alak*
`exactly(3,[<X>,2],1)`	`exactly(3,[<X = 1..3>,2],1)`
`X = 1..3 -> {3}`	`X = [1,2,3] -> [3]`
`Constraint exited.`	`Constraint exited.`

6.7. Testreszabás saját megjelenítővel

- ***my_global_visualizer*(+*Arg1*, +*Arg2*, …, *Constraint*, *Actions*)**
  my_global_visualizer helyére tetszőleges predikátumnév kerülhet, *Arg1*, *Arg2* …a predikátum által használt saját argumentumok, *Constraint* és *Actions* pedig az a két paraméter, amit az `fdbg_show/2` rak a paraméterlista végére. *Constraint* az éppen felébredt korlátot, *Actions* pedig a korlát által visszaadott akciólistát tartalmazza. A korlát-megjelenítő nevét és paraméterlistáját (az utolsó kettő kivételével) az `fdbg_on` opciólistájában kell átadni:
  `fdbg_on(constraint_hook(`*my_global_visualizer*`(`*Arg1*`, `*Arg2*`, …)))`.

- ***my_labeling_visualizer*(+*Arg1*, +*Arg2*, …, *Event*, *ID*, *Var*)**
  Event a megjelenítő meghívását kiváltó címkézési esemény, a következőek szerint:

  - `start` — címkézés kezdete
  - `fail` — címkézés meghiúsulása
  - `step(`*Step*`)` — címkézési lépés, amelyet *Step* ír le

  ID a címkézés azonosítója, *Var* pedig a címkézett változó. A címkézés-megjelenítő nevét és paraméterlistáját (az utolsó kettő kivételével) az `fdbg_on` opciólistájában kell átadni:
  `fdbg_on(labeling_hook(`*my_labeling_visualizer*`(`*Arg1*`, `*Arg2*`, …)))`.

Érdemes egy pillantást vetni az `fdbg_show/2` beépített megjelenítő kódjára is, mert ez irányelvet adhat saját megjelenítő írásához:

```
fdbg_show(Constraint, Actions) :-
        fdbg_annotate(Constraint, Actions, AnnotC, CVars),
        print(fdbg_output, AnnotC),
        nl(fdbg_output),
        fdbg_legend(CVars, Actions),
        nl(fdbg_output).
```

Gyakran szükség lehet arra, hogy csak bizonyos korlátok működését vizsgáljuk, ilyenkor jön jól egy saját `fdbg_show/2` eljárás:

```
filtered_show(Constraint, Actions) :-
        Constraint = scalar_product(_,_,_,_),
        fdbg_show(Constraint, Actions).
```

A fenti predikátum meghiúsulhat, ha `Constraint` nem `scalar_product/4` korlát, de a megjelenítők esetében a meghiúsulás nem okozza automatikusan az egész Prolog végrehajtás meghiúsulását, ezért az ilyen jellegű megoldás megengedett. A megjelenítő használata:

```
:- fdbg_on([constraint_hook(filtered_show), file('fdbg.log', write)]).
```

6.8. Egyéb segéd-predikátumok

A változók tartományának kiírásához és az úgynevezett *annotáláshoz* több predikátum adott. Ezeket használják a beépített nyomkövetők, de hívhatók kívülről is. Ebben az alfejezetben ezeket a predikátumokat ismertetjük.

- **`fdbg_annotate(+`*`Term0`*`, -`*`Term`*`, -`*`Vars`*`)`**
  `fdbg_annotate(+`*`Term0`*`, +`*`Actions`*`, -`*`Term`*`, -`*`Vars`*`)`
  A *`Term0`* kifejezésben található összes FD változót megjelöli, azaz lecseréli egy `fdvar/3` struktúrára. Ennek tartalma:

  - a változó neve
  - a változó maga (tartománya még a szűkítés előtti állapotokat tükrözi)
  - egy FD halmaz, amely a változó tartománya *lesz* az *`Actions`* akciólista szűkítései után.

  Az így kapott kifejezés lesz *`Term`*, a beszúrt `fdvar/3` struktúrák listája pedig *`Vars`*.

  Példa az `fdbg_annotate/3` használatára:

  ```
  | ?- length(L, 2), domain(L, 0, 10), fdbg_assign_name(L, x),
       L=[X1,X2], fdbg_annotate(lseq(X1,X2), Goal, _),
       format('write(Goal) --> ~w~n', [Goal]),
       format('print(Goal) --> ~p~n', [Goal]).

  write(Goal) --> lseq(fdvar(x_1,_2,[[0|10]]),fdvar(x_2,_2,[[0|10]]))
  print(Goal) --> lseq(<x_1>,<x_2>)
  ```

Látható, hogy az `fdvar/3` struktúrákra az `fdbg` modul definiál egy saját `portray/1` megjelenítő eljárást, ezért ha a `print/1` használatával íratjuk ki az ilyen struktúrákat, akkor azok a fenti tömör módon jelennek meg.

- **`fdbg_legend(+`*`Vars`*`)`**
  `fdbg_legend(+`*`Vars`*`, +`*`Actions`*`)`
  Az `fdbg_annotate/3,4` használatával előálló *`Vars`* változólistát és az *`Actions`* akció-listából levonható következtetéseket jelmagyarázatként kiírja a következő szabályok szem előtt tartásával:
  - egy sorba egy változó leírása kerül
  - minden sor elején a változó neve szerepel
  - a nevet a változó tartománya követi (régi `->` új)

6.9. A mágikus sorozatok feladat nyomkövetése

Ebben az alfejezetben a mágikus sorozatok feladat egy lehetséges megoldásán mutatjuk be az `fdbg` nyomkövető csomag kimenetét. A programkódban csak a nyomkövetés megértéséhez szükséges predikátumokat tüntettük fel:

```
magic(N, L) :-
        length(L, N),
        fdbg_assign_name(L, x), % <--- !!!
        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).

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

A kimenet vége, az utolsó címkézési lépés után:

```
exactly(0,[1,2,<x_3>,<x_4>],1)          x_3 = 0..3
                                        x_4 = 0..3

exactly(2,[1,2,<x_3>,<x_4>],<x_3>)      x_3 = 0..3 -> 1..3
                                        x_4 = 0..3

exactly(3,[1,2,<x_3>,<x_4>],<x_4>)      x_3 = 1..3
                                        x_4 = 0..3 -> 0..2

exactly(1,[1,2,<x_3>,<x_4>],2)          x_3 = 1..3
                                        x_4 = 0..2
```

```
exactly(2,[1,2,<x_3>,<x_4>],<x_3>)        x_3 = 1..3
                                          x_4 = 0..2

exactly(0,[1,2,<x_3>,<x_4>],1)            x_3 = 1..3
                                          x_4 = 0..2 -> {0}
                                          Constraint exited.

exactly(1,[1,2,<x_3>,0],2)                x_3 = 1..3 -> {1}
                                          Constraint exited.

exactly(2,[1,2,1,0],1)                    Constraint exited.

exactly(3,[1,2,1,0],0)                    Constraint exited.

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

7. fejezet

Esettanulmányok `clpfd`-ben

Ebben a fejezetben néhány nagyobb `clpfd` feladat megoldását ismertetjük, közben pedig bemutatjuk a *konstruktív diszjunkció*, a *duális címkézés* és a *borotválás* módszerét.

7.1. Négyzetdarabolás

Adott egy nagy négyzet oldalhosszúsága (pl. `Limit = 10`), valamint adottak kis négyzetek oldalhosszúságai (pl. `Sizes = [6,4,4,4,2,2,2,2]`). A kis négyzetek területösszege megegyezik a nagy négyzet területével. Meg kell határozni, hogy le lehet-e fedni a kis négyzetekkel a nagy négyzetet, és ha igen, meg is kell adni azt, hogy a lefedéshez hova kell helyezni a kis négyzeteket (a nagy négyzet bal alsó sarka az (1,1) koordinátán van). A példában említett feladat megoldása: `Xs = [1,7,7,1,5,5,7,9]`, `Ys = [1,1,5,7,7,9,9,9]`.

Források:

- Pascal van Hentenryck et al. tanulmányának 2. szekciója (`http://www.cs.brown.edu/publications/techreports/reports/CS-93-02.html`)
- SICStus `clpfd` példaprogram: `library('clpfd/examples/squares')`

Néhány tesztadat:

Limit	Sizes
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]
503	[211,179,167,157,149,143,135,113,100,93,88,87,67,62,50,34,33,27, 25,23,22,19,16,15,4]

Az esettanulmány program-változatai, adatai, tesztkörnyezete megtalálható itt: `http://www.cs.bme.hu/~szeredi/oktatas/nlp/nlp_progs_sq.tgz`.

A megoldás során nem foglalkozunk az azonos oldalhosszak miatt jelentkező többszörös megoldások kiküszöbölésével, mivel az eredeti feladat különböző oldalhosszúságú négyzetekről szólt, az azonos oldalhosszak csak azért kerültek bele, hogy a programot kisebb tesztadatokon is tesztelni tudjuk. A teszteseteket minden alkalommal Linux operációs rendszer alatt egy 600 MHz-es Pentium III gépen futtattuk maximum 120 másodpercig. Ahol a programvariáns nem adott eredményt 120 másodpercen belül, ott a futási táblázatokban a mezőt üresen hagytuk. A táblázatokban a futási időt és a visszalépések számát is feltüntetjük.

7.1.1. Egyszerű Prolog megoldás

Az alábbi program Colmerauer `clpr` megoldásán alapul, a működési elve hasonlít a 24. oldalon található téglalap-lefedő feladathoz.

```
% A Limit méretű négyzet lefedhető diszjunkt, Ss oldalhosszúságú
% négyzetekkel, amelyek X és Y kooridnátáit az Xs és Ys lista tartalmazza
squares_prolog(Ss, Limit, Xs, Ys) :-
        triples(Ss, Xs, Ys, SXYs),
        Y0 is Limit+1,
        XY0 = 1-Y0,
        NLimit is -Limit,
        filled_hole([NLimit,Limit,Limit], _, XY0, SXYs, []).

% triples(Ss, Xs, Ys, SXYs): SXYs is s(S,X,Y) alakú struktúrákból álló lista
triples([S|Ss], [X|Xs], [Y|Ys], [s(S,X,Y)|SXYs]) :-
        triples(Ss, Xs, Ys, SXYs).
triples([], [], [], []).

% filled_hole(L0, L, XY, SXYs0, SXYs): az L0 vonalban lévő, XY pontban
% kezdődő lyuk az SXYs0 és SXYs listák különbségében lévő négyzetekkel
% lefedve az L vonalat adja
filled_hole(L, L, _, SXYs, SXYs) :-
        L = [V|_], V >= 0, !.
filled_hole([V|HL], L, X0-Y0, SXYs00, SXYs) :-
        V < 0 , Y1 is Y0+V,
        select(s(S,X0,Y1), SXYs00, SXYs0),
        placed_square(S, HL, L1),
        Y2 is Y1+S, X2 is X0+S,
        filled_hole(L1, L2, X2-Y2, SXYs0, SXYs1),
        V1 is V+S,
        filled_hole([V1,S|L2], L, X0-Y0, SXYs1, SXYs).

% placed_square(S, HL, L): a HL vízszintes vonalon az S méretű négyzetet
% elhelyezve az L függőleges vonalat kapjuk
placed_square(S, [H,0,H1|L], L1) :-
        S > H, !, H2 is H+H1,
        placed_square(S, [H2|L], L1).
placed_square(S, [H,V|L], [X|L]) :-
        S = H, !, X is V-S.
placed_square(S, [H|L], [X,Y|L]) :-
        S < H, X is -S, Y is H-S.
```

variáns	10		20		112		175		503	
Prolog	0.000	0	0.87	271K	0.38	183K	5.72	2.6M	93.58	29M

7.1.2. Egyszerű `clpfd` megoldás

Ez a megoldás veszi a kis négyzetek összes koordinátáját, és beállítja őket úgy, hogy értéküket a nagy négyzeten belül vegyék fel. Ezután minden négyzet-párra felveszi a `no_overlap/6` constraintet, ami azt írja le Prolog választási pontok segítségével, hogy a két négyzet nem fedi egymást. Végül címkézéssel megadja az eredményt.

```
% A feladatra adott egyszerű megoldás spekulatív diszjunkció használatával
squares_spec(Sizes, Limit, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Limit),
        state_asymmetry(Xs, Ys, Sizes, Limit),
        state_no_overlap(Xs, Ys, Sizes),
        labeling([], Xs), labeling([], Ys).

% Legenerálja a koordinátákra vonatkozó határokat
generate_coordinates([], [], [], _).
generate_coordinates([X|Xs], [Y|Ys], [S|Ss], Limit) :-
        Sd is Limit-S+1, domain([X,Y], 1, Sd),
        generate_coordinates(Xs, Ys, Ss, Limit).

% Az első négyzet középpontja a bal alsó negyedben van,
% a főátló alatt
state_asymmetry([X|_], [Y|_], [D|_], Limit) :-
        UB is (Limit-D+2)>>1, X in 1..UB, Y #=< X.

% Páronkénti át nem fedést biztosító korlátok felvétele
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).

% Megadja, hogy az (X,Y) középpontú, S méretű négyzet nem fedi át a
% többit, amiket a listákban adunk át
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).
state_no_overlap(_, _, _, [], [], []).

% no_overlap_spec(X1,Y1,S1, X2,Y2,S2):
% Az SQ1 = <X1,Y1,S1> négyzet nem fedi át SQ2 = <X2,Y2,S2> -t
% Spekulatív megoldás
no_overlap_spec(X1, _Y1, _S1, X2, _Y2, S2) :-
        X2+S2 #=< X1.   % SQ1 is above SQ2
no_overlap_spec(X1, _Y1, S1, X2, _Y2, _S2) :-
        X1+S1 #=< X2.   % SQ1 is below SQ2
no_overlap_spec(_X1, Y1, _S1, _X2, Y2, S2) :-
        Y2+S2 #=< Y1.   % SQ1 is to the right of SQ2
no_overlap_spec(_X1, Y1, S1, _X2, Y2, _S2) :-
        Y1+S1 #=< Y2.   % SQ1 is to the left of SQ2
```

Ezzel a megoldással sokkal kevésbé hatékony változatot kaptunk:

variáns	10		20		112		175		503	
Prolog	0.000	0	0.87	271K	0.38	183K	5.72	2.6M	93.58	29M
`spec`	1.99	34K								

Érdemes megfigyelni, hogy míg a Prolog megoldás a 10-es feladatot visszalépés nélkül oldotta meg, addig ehhez a `spec` változatnak 34 ezer visszalépésre volt szüksége. A problémát a `no_overlap/6` korlát spekulatív diszjunkcióval való megvalósítása jelenti, hiszen ez a kombinatorikus robbanás miatt nagyon sok visszalépést eredményez.

7.1.3. A diszjunkció megvalósítási módszerei

Az előző variánsban gondot okozó spekulatív diszjunkció kiváltására több lehetőségünk is van, ezeket egy egyszerűbb példán fogjuk megnézni. Legyen ez az egyszerűbb példa a következő:

```
| ?- domain([X,Y], 0, 6), ( X+5 #=< Y ; Y+5 #=< X).
X in 0..1, Y in 5..6 ? ;
X in 5..6, Y in 0..1 ? ;
no
```

A diszjunkció megvalósítására kézenfekvő megoldás, ha tükrözést használunk:

```
| ?- domain([X,Y], 0, 6), X+5 #=< Y #\/ Y+5 #=< X.
X in 0..6, Y in 0..6 ? ;
no
```

Amint látjuk, ennek az a hátránya, hogy nem hajt végre maximális szűkítést, nem jön rá, hogy `X` és `Y` semmiképpen nem lehet 2, 3 vagy 4. A korlát-megoldó ugyanis egy diszjunkció esetén csak akkor tud tenni valamit, ha a diszjunkcióban részt vevő korlátok közül egyet kivéve már az összesről eldőlt, hogy nem állhat fenn. Érdemes tehát további megoldásokon gondolkoznunk. A jelen esetben például kikerülhetjük a diszjunkciót az `abs` korlát használatával:

```
| ?- domain([X,Y], 0, 6), 'x+y=t tsz'(Y, D, X), abs(D) #>= 5.
X in(0..1)\/(5..6), Y in(0..1)\/(5..6) ? ;
no
```

Ezt azonban nagyon sok esetben nem tehetjük meg. Átírhatjuk viszont a diszjunkciót indexikálissá:

```
ix_disj(X, Y) +:
        X in \(max(Y)-4..min(Y)+4), Y in \(max(X)-4..min(X)+4).

| ?- ix_disj(X, Y).
X in(0..1)\/(5..6), Y in(0..1)\/(5..6) ? ;
no
```

Sajnos az indexikálisokra fennálló korlátozások (pl. fix számú változó) miatt ezt is csak speciális esetekben tehetjük meg. Most egy általános szűkítési módszert, a *konstruktív diszjunkciót* fogjuk ismertetni.

A konstruktív diszjunkció alapötlete a következő: a diszjunkció minden tagja esetén vizsgáljuk meg a hatását a tárra, és jelöljük az így kapott „vagylagos” tárakat $S_1, \ldots, S_n$-nel. Ekkor minden változó a vagylagos tárakban kapott tartományok uniójára szűkíthető: `X in_set` $\cup D(\mathtt{X}, S_i)$. A konstruktív diszjunkciót általánosan az alábbihoz hasonló módon lehet megvalósítani:

```
cdisj(Cs, Var) :-
        empty_fdset(S0), cdisj(Cs, Var, S0, S),
        Var in_set S.

cdisj([Constraint|Cs], Var, Set0, Set) :-
        findall(S, (Constraint,fd_set(Var,S)), Sets),
        fdset_union([Set0|Sets], Set1),
        cdisj(Cs, Var, Set1, Set).
cdisj([], _, Set, Set).

| ?- domain([X,Y], 0, 6), cdisj([X+5 #=< Y,Y+5 #=< X], X).
X in(0..1)\/(5..6), Y in 0..6 ?
```

A konstruktív diszjunkció akár erősebb is lehet, mint a tartomány-szűkítés, mert más korlátok hatását is figyelembe tudja venni, lásd az alábbi példát:

```
| ?- domain([X,Y], 0, 20), X+Y #= 20, cdisj([X#=<5,Y#=<5],X).
X in(0..5)\/(15..20), Y in(0..5)\/(15..20) ?
```

7.1.4. `clpfd` megvalósítás reifikációval és indexikálissal

Számosság-alapú `no_overlap` változatok

```
no_overlap_card1(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.

no_overlap_card2(X1, Y1, S1, X2, Y2, S2) :-
  call( abs(2*(X1-X2)+(S1-S2)) #>= S1+S2 #\/
        abs(2*(Y1-Y2)+(S1-S2)) #>= S1+S2 ).
```

Indexikális `no_overlap` („gyenge” konstruktív diszjunkció)

Alapgondolat: Ha két négyzet Y irányú vetületei biztosan átfedik egymást, akkor X irányú vetületeik diszjunktak kell legyenek, és fordítva. Az Y irányú vetületek átfedik egymást, ha mindkét négyzet felső széle magasabban van mint a másik négyzet alsó széle: `Y1+S1>Y2` és `Y2+S2>Y1`. Indexikálisban ezt a következőképpen tudjuk megfogalmazni: ha a `(Y1+S1..Y2) \/ (Y2+S2..Y1)` halmaz üres, akkor a fenti feltétel fennáll, tehát X irányban szűkíthetünk: `X1 =< X2-S1` vagy `X1 >= X2+S2`. Feltételes kifejezéssel:

```
X1 in ((Y1+S1..Y2)\/(Y2+S2..Y1))?(inf..sup) \/ \(X2-S1+1..X2+S2-1)
```

Ezen ötlet felhasználásával az indexikális:

```
no_overlap_ix(X1, Y1, S1, X2, Y2, S2) +:
%       ha Y irányú átfedés van, azaz
%       ha min(Y1)+S1 > max(Y2) és min(Y2)+S2 > max(Y1) ...
        X1 in ((min(Y1)+S1..max(Y2)) \/ (min(Y2)+S2..max(Y1)))
%                       ... akkor X irányban nincs átfedés:
             ? (inf..sup) \/ \(max(X2)-(S1-1) .. min(X2)+(S2-1)),
        X2 in ((min(Y1)+S1..max(Y2)) \/ (min(Y2)+S2..max(Y1)))
             ? (inf..sup) \/ \(max(X1)-(S2-1) .. min(X1)+(S1-1)),
        Y1 in ((min(X1)+S1..max(X2)) \/ (min(X2)+S2..max(X1)))
             ? (inf..sup) \/ \(max(Y2)-(S1-1) .. min(Y2)+(S2-1)),
        Y2 in ((min(X1)+S1..max(X2)) \/ (min(X2)+S2..max(X1)))
             ? (inf..sup)\/ \(max(Y1)-(S2-1) .. min(Y1)+(S1-1)).
```

variáns	10		20		112		175		503	
Prolog	0.00	0	0.87	271K	0.38	183K	5.72	2.6M	93.58	29M
`spec`	1.99	34K								
`card1`	0.07	141								
`card2`	0.07	141								
`ix`	0.01	141								

A visszalépések száma lényegesen javult a spekulatív diszjunkcióval kapott változathoz képest, azonban a program még így is nyomába sem ér a hagyományos Prolog megvalósításnak.

7.1.5. Kapacitás-korlátok és paraméterezhető címkézés

Mint azt már említettük, lehetőség van a címkézés menetének befolyásolására a `labeling/2` eljárás paraméterlistáján keresztül. Az eredeti Prolog megoldás a „tetris-elv” szerint alulról felfelé töltötte fel a nagy négyzetet a kis négyzetekkel, ennek egy elég jó megközelítése a `[min,step]` üzemmódú címkézés. Próbálgatás céljából esetleg érdemes külön paraméterezhetővé tenni a címkézési módokat, majd összevetni az egyes variánsokat.

További gyorsítás érhető el redundáns korlátok használatával. A jelenlegi program ugyanis még nem elég okos: például amikor a nagy négyzet alja betelt, nem hagyja ki az Y változók tartományából az 1 értéket, pedig oda már biztosan nem tudunk további négyzeteket rakni. Az ún. kapacitás-korlátokkal ez megvalósítható: ha összeadjuk azon kis négyzetek oldalhosszát, amelyek elmetszenek egy X=1, X=2, ..., Y=1, Y=2, ...vonalat, akkor a nagy négyzet oldalhosszát kell kapnunk (a kis négyzeteket itt alulról és balról zártnak, felülről és jobbról nyíltnak tekintjük). Például X irányban:

$$\sum\{\mathtt{S}_i | p \in [\mathtt{X}_i, \mathtt{X}_i + \mathtt{S}_i)\} = \mathtt{Limit} \quad (\forall p \in \mathtt{1..Limit-1})$$

A kapacitás-korlátok az alábbi kódrészlettel felvehetőek:

```
% A feladatra adott egyszerű megoldás kapacitás-korlátok használatával
squares_cap(Lab, Sizes, Limit, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Limit),
        state_asymmetry(Xs, Ys, Sizes, Limit),
```

```
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(1, Xs, Sizes, Limit),
        state_capacity(1, Ys, Sizes, Limit),
        labeling(Lab, Xs), labeling(Lab, Ys).

% Kapacitás-korlát a Cs koordinátákra Sizes oldalhosszúságú
% kis négyzetek és Limit méretű nagy négyzet esetén a
% Pos..Limit intervallum összes elemére
state_capacity(Pos, Limit, Cs, Sizes) :-
        Pos =< Limit, !, accumulate(Cs, Sizes, Pos, Bs),
        scalar_product(Sizes, Bs, #=, Limit),
        Pos1 is Pos+1, state_capacity(Pos1, Limit, Cs, Sizes).
state_capacity(_Pos, _Limit, _, _).

% accumulate(C, S, Pos, B): B, C és S ugyanolyan hosszú listák,
% B_i-k B elemei, B_i = 1 ⇔ Pos ∈ [C_i, C_i + S_i),
accumulate([], [], _, []).
accumulate([Ci|Cs], [Si|Ss], Pos, [Bi|Bs]) :-
        Crutch is Pos-Si+1, Ci in Crutch .. Pos #<=> Bi,
        accumulate(Cs, Ss, Pos, Bs).
```

variáns, címkézés	10		20		112		175		503	
Prolog	0.00	0	0.87	271K	0.38	183K	5.72	2.6M	93.58	29M
`[]-ix, [min]`	0.01	84								
`cap-ix, []`	0.01	0	0.07	18						
`cap-ix, [min]`	0.01	0	0.06	0	1.96	109	3.74	105	20.32	405
`cap-spec, [min]`	2.31	34K								
`cap-card1, [min]`	0.04	0	0.24	0	3.51	109	4.86	105	22.63	405
`cap-card2, [min]`	0.04	0	0.34	0	2.41	109	4.48	105	21.83	405

Amint látható, a kapacitás-korlátokkal a nagyobb méretekre a `clpfd` megoldás már lekörözi a hagyományos Prolog megoldást.

7.1.6. Ütemezési és lefedési korlátok használata

A négyzetdarabolás felfogható ütemezési problémaként, illetve diszjunkt téglalapok problémájaként is. Mindkét feladatra van beépített korlát a SICStusban. Ütemezési probléma esetén alkalmazhatjuk a `cumulative/5` korlátot mindkét tengely irányában, diszjunkt téglalapok problémája esetén pedig a `disjoint2/2` korlátot (ilyenkor a `no_overlap` használatától akár el is tekinthetünk).

A két újabb változat:

```
squares_cum(Lab, Opts, Sizes, Limit, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Limit),
        state_asymmetry(Xs, Ys, Sizes, Limit),
        state_no_overlap(Xs, Ys, Sizes),
        cumulative(Xs, Sizes, Sizes, Limit, Opts),
        cumulative(Ys, Sizes, Sizes, Limit, Opts),
        labeling(Lab, Xs), labeling(Lab, Ys).
```

```
squares_dis(Lab, Opts, Sizes, Limit, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Limit),
        state_asymmetry(Xs, Ys, Sizes, Limit),
        state_no_overlap(Xs, Ys, Sizes),      % ez elmarad a`none'
                                              % variáns esetén
        disjoint2_data(Xs, Ys, Sizes, Rects),
        disjoint2(Rects, Opts),
        labeling(Lab, Xs), labeling(Lab, Ys).

disjoint2_data([], [], [], []).
disjoint2_data([X|Xs], [Y|Ys], [S|Ss], [r(X,S,Y,S)|Rects]) :-
        disjoint2_data(Xs, Ys, Ss, Rects).
```

Az alábbi tesztek során mindig `[min]` címkézést használtunk, és a globális korlátok paraméterezésével variáltunk. Rövidítések: `e = edge_finder(true)`, `g = global(true)`.

variáns	10		20		112		175		503	
`cum-ix`	0.00	0	0.02	0						
`cum(e)-ix`	0.01	0	0.01	0	0.18	139	0.12	67	0.52	421
`dis-none`	0.01	52								
`dis(g)-none`	0.00	0	0.01	0	0.73	282	0.41	133	2.55	576
`dis(g)-ix`	0.00	0	0.02	0	0.93	282	0.53	133	2.95	576

7.1.7. Duális címkézés

Duális címkézés során nem a változókhoz keresünk megfelelő értéket, hanem az értékekhez megfelelő változót. Kicsit formálisabban fogalmazva az algoritmus lényege:

- vegyük sorra a lehetséges változó-értékeket,
- egy adott e értékhez keresünk egy V változót, amely felveheti ezt az értéket,
- csináljunk egy választási pontot: $V = e$, vagy $V \neq e$, stb.

Növekvő értéksorrend esetén a keresési tér meg fog egyezni a `[min, step]` beépített címkézés keresési terével.

```
% dual_labeling(L, Min, Max): címkézi az L lista elemeit, ahol
% minden L-beli X változóra X in Min..Max fennáll.
% A programban használt hívási formátum:
% dual_labeling(Xs,1,Limit),dual_labeling(Ys,1,Limit).
dual_labeling([], _, _) :- !.
dual_labeling(L0, Min0, Limit) :-
        dual_labeling(L0, L1, Min0, Limit, Min1),
        dual_labeling(L1, Min1, Limit).

% dual_labeling(L0, L, I, Min0, Min): címkézi az L0 lista változóit
% az I értékkel, amikor ez lehetséges, a maradék változókat L-ben adja
% vissza. Ugyanakkor Min0-Min-ben gyűjti az L-beli változók
```

```
% alsó határát
dual_labeling([], [], _, Min, Min).
dual_labeling([X|L0], L, I, Min0, Min) :-
        (   integer(X) -> dual_labeling(L0, L, I, Min0, Min)
        ;   X = I,
            dual_labeling(L0, L, I, Min0, Min)
        ;   X #> I,
            fd_min(X, Min1), Min2 is min(Min0,Min1),
            L = [X|L1], dual_labeling(L0, L1, I, Min2, Min)
        ).
```

variáns; címkézés	10		20		112		175		503	
`cum(e)-ix; [min]`	0.01	0	0.01	0	0.18	139	0.12	67	0.52	421
`cum(e)-ix; dual`	0.01	0	0.02	0	0.19	139	0.13	67	0.54	421
`cap-cum(e)-ix;`	0.02	0	0.07	0	1.77	100	3.22	65	17.26	395
`cap-dis(g)-none;`	0.01	0	0.06	0	1.71	97	3.24	66	17.98	393
`cum(e),dis(g)-none;`	0.00	0	0.01	0	0.23	136	0.16	67	0.99	419

7.2. Torpedó

Adott egy téglalap alakú táblázat, amelyben $1 \times n$-es hajókat kell elhelyezni úgy, hogy még átlósan se érintkezzenek. 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 adottak az ismert torpedó-mentes (tenger) mezők.

Példa: Két szín, mindkét színből 1 darab egyes és 1 darab kettes hajó. Ismert mezők: az 1. sor 1. mezője tenger, az első sor 3. mezője egy kettes hajó tatja (jobb vége).

```
      A feladat:                                  A megoldás:

      1 2 3 4 5     <--  oszlopszám                1 2 3 4 5
      0 1 1 1 0     <--  1. oszlopössz.            0 1 1 1 0

1  2  =   r      0                              1  2  = * r : :  0
2  0             1                              2  0  : : : : #  1
3  0             1                              3  0  # : : : :  1
4  1             1                              4  1  # : : * :  1
   ^-------------^------ sorösszegek
      2 0 0 0 1     <--  2. oszlopössz.            2 0 0 0 1
```

A fenti példában alkalmazott jelölésrendszer: a tábla felett az első számsor az oszlopok számozását jelenti, a tábla mellett az első oszlop a sorok számozását. Közvetlenül a tábla szélei mellett lévő számsorok az adott sorban, illetve oszlopban lévő hajódarabkák számát adják meg, a felső, illetve a bal oldali az 1. színét, az alsó, illetve a jobb oldali pedig a 2. színét. A táblában a tengert = (egyenlőségjel) karakterrel, a hajódarabkákat pedig betűkkel jelöljük. Az 1. szín hajódarabkáit kisbetűkkel, a 2. színét nagybetűkkel ábrázoljuk. Az 1 hosszú hajók `o`, illetve `O` betűkkel vannak jelölve, a hosszabb hajók esetén pedig `u` (`U`) a hajó felső vége, `d` (`D`) az alsó vége, `l` (`L`) a bal széle, `r` (`R`) a jobb széle, `m` (`M`) pedig a közepe. A kikövetkeztetett hajódarabkákat `*` (csillag) és `#` (hashmark) karakterek jelölik, a kikövetkeztetett tengerdarabkákat pedig `:` (kettőspont).

A feladat az 1999. évi NLP kurzus nagyházifeladata volt, a mintamegoldás letölthető az alábbi címről: `http://www.cs.bme.hu/~szeredi/oktatas/nlp/hf_99_torpedo.tgz`.

7.2.1. A feladat modellezése

A feladat komplexitásából kifolyólag eleve érdemes azon elgondolkoznunk, hogy hogyan feleltessük meg a feladatot a CLP világnak. Például a korlát-változók felvételére is eleve két lehetőségünk van:

a. Minden hajóhoz hozzárendelünk egy irányváltozót (vízszintes vagy függőleges) és a kezdőpont koordinátáit. Így viszonylag kevés változóval „megússzuk”, cserébe viszont szimmetria problémák léphetnek fel (azonos méretű hajók sorrendje), a korlátjaink bonyolultabbak lesznek és sok diszjunktív korlátot kell alkalmaznunk (pl. egy hajó vízszintes vagy függőleges elhelyezés esetén más-más mezőket fed le).

b. A mezőkhöz rendelünk változókat, és mezőnként tároljuk, hogy mi található ott: hajó-darab vagy tenger. Ezzel ugyan több változónk lesz, de a korlátok lényegesen leegyszerűsödnek, ezért ezt a megoldást választjuk.

Ezek után el kell gondolkoznunk azon, hogy az egyes mezőkhöz tartozó változóknak milyen értékkészletet adunk. Újfent két, lényegében eltérő megoldás között választhatunk:

a. egy mezőről csak azt tároljuk, hogy hajódarab vagy pedig tenger van ott, a hajódarabról pedig a színt is megjegyezzük. Mivel nem rögzítjük, hogy egy hajódarab a hajó melyik részét alkotja, ezért az eleve ismert mezőknél információvesztés lép fel.

b. az egyes hajódarabokat is megkülönböztetjük:

 b1. az előre kitöltött mezőknek megfelelő darabok `(u,l,m,r,d,o)` — ilyenkor ismét diszjunktív korlátokat kell alkalmazni (pl. ugyanaz a betű többféle hajó része lehet)

 b2. részletesebb bontás: a mezőket megkülönböztetjük a hajó hossza, iránya, a darab hajón belüli pozíciója szerint, pl.: egy 4 hosszú vízszintes hajó balról 3. darabja. A megoldásban ezt a módszert alkalmazzuk.

Vegyük észre, hogy a választott megoldás jellemzője az, hogy ha egy mezőhöz tartozó változó tengertől különböző értéket kap, akkor ezzel már az egész hajót meghatároztuk.

Mivel egy mezővel kapcsolatban több információt is rögzíteni akarunk, felvetődik a kérdés, hogy külön változókkal adjuk meg az egyes jellemzőket, vagy pedig egyetlen változóban jelenítsük meg az összeset. Az első esetben nyilvánvalóan egyszerűbb lesz a kódolás, de a korlátok szűkítései gyengébbek lesznek, mivel egy korlátnak nem feltétlenül áll rendelkezésére az összes információ. A második esetben ugyan bonyolultabb kódolást kell megvalósítani, de cserébe a korlátok erősebben szűkítenek majd, és mivel a megoldásunkban elsősorban a sebesség a lényeg, ezért ehhez a módszerhez érdemes folyamodni.

Összefoglalva a választott irányelveket:

- Minden mezőnek egy változó felel meg.
- Az értékek kódolási elvei (`max` címkézéshez igazítva)
  - 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 = horizontális (vízszintes), i hosszú hajó j-edik darabja, vij = vertikális (függőleges) hajó megfelelő darabja, stb. A kódkiosztás:

```
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
```

7.2.2. Alapvető korlátok

A feladat megoldásához két alapvető korlátra lesz szükségünk:

- `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} ? ;
no
```

- `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. Ez a `count/4` eljárás kiterjesztése: nem egyetlen szám, hanem egy számhalmaz előfordulásait számoljuk meg.

Ezen korlátokat az ismert mezők megfelelő csoportokra való megszorítása után a következőképpen kell felvenni:

1. Színenként az adott sor- és oszlopszámlálók előírása (erre jó az előző részfejezetben felvázolt `group_count/4` predikátum).
2. A hajóorr-darabok megszámolásával az adott hajófajta darabszámának biztosítása (`group_count/4`, minden színre és minden hajófajtára).
3. A vízszintes, függőleges és átlós irányú szomszédos mezőkre vonatkozó korlátok biztosítása a `coded_field_neighbour/3` korláttal.

A 2. fajtájú korlátoknál a részösszegekre néhol érdemes segédváltozókat bevezetni (pl. `A+B+C #= 2, A+B+D #= 2` helyett `A+B #= S, S+C #= 2, S+D #= 2` jobban tud szűkíteni, mert az `S` változón keresztül a két összegkorlát „kommunikál” egymással). Formálisan: jelölje sor_s^K ill. $oszl_s^L$ az s hajódarab előfordulási számát a K-adik sorban, ill. az L-edik oszlopban. A hajók számolásához a $sor_{\mathtt{hI1}}^K$ és $oszl_{\mathtt{vI1}}^L$ mennyiségekre segédváltozókat vezetünk be, ezekkel a 3. korlát:

$$\text{az } \mathtt{I} \text{ hosszú hajók száma} = \textstyle\sum_K sor_{\mathtt{hI1}}^K + \sum_L oszl_{\mathtt{vI1}}^L \qquad (\mathtt{I} > 1)$$

$$\text{az 1 hosszú hajók száma} = \textstyle\sum_K sor_{\mathtt{h11}}^K$$

7.2.3. Redundáns korlátok, címkézés és borotválás

A fenti alapvető korlátok mellé még az alábbi redundáns korlátokat érdemes bevezetni:

- `count_ships_occs`: sorösszegek alternatív kiszámolása (vö. a mágikus sorozatok megoldásában a skalárszorzat redundáns korláttal):

$$\text{a } K\text{. sorbeli darabok száma} = \sum_{\mathtt{I} \leq hosszak} \mathtt{I}{*}sor_{\mathtt{hI1}}^K + \sum_{1<\mathtt{I} \leq hosszak, \mathtt{J} \leq \mathtt{I}} sor_{\mathtt{vIJ}}^K$$

  Analóg módon az oszlopösszegekre is.

  Ennek a korlátnak a hatására „veszi észre” a program, hogy ha pl. egy sorösszeg 3, akkor nem lehet a sorban 3 eleműnél hosszabb hajó.

- `count_ones_columns`: az egy hosszú darabok számát az oszloponkénti előfordulások összegeként is meghatározzuk.

- `count_empties`: minden sorra és oszlopra a tenger-mezők számát is előírjuk (a sorhosszból kivonva az összes — különböző színű — hajódarab összegét).

A mintamegoldásban az alábbi címkézési fajták vannak implementálva (`label(`*`Variáns`*`)` opciók):

- `plain`: `labeling([max,down], Mezők)`, ahol `Mezők` a mezőváltozókat tartalmazó lista.
- `max_dual`: a négyzetkirakáshoz hasonlóan a legmagasabb *értékeket* próbálja a változóknak értékül adni. Ez szűkítő hatásban (és így a keresési fa szerkezetében) azonos a `plain` variánssal.
- `ships`: speciális címkézés, minden hosszra, a legnagyobbtól kezdve, minden színre az adott színű és hosszú hajókat sorra elhelyezi (ez az alapértelmezés).

A megoldás a konstruktív diszjunkció egy egyszerűsített verzióját, a *borotválás*t is használja a címkézés során. A borotválás lényege az, hogy minden n. címkézési lépésben a címkézésből hátralévő változók mindegyikét megpróbáljuk egy adott tartományra (jelen esetben „tenger”-re) helyettesíteni, és ha ez azonnal meghiúsulást okoz, akkor a kipróbált tartományt kizárhatjuk a változó tartományából (jelen esetben megállapíthatjuk, hogy azon a mezőn hajódarab van). A módszert n változtatásával és a tartomány megválasztásával lehet „finomhangolni”. A torpedó feladatban alkalmazott borotválást minden szín címkézése előtt megismételjük. A `filter(`*`VariánsLista`*`)` opción keresztül változtathatjuk a borotválás jellegét: ha a lista eleme `off`, akkor nincs borotválás, `on` esetén egyszeres borotválás van, repetitive esetén pedig minden borotválásnál ismételten szűrünk addig, amíg az újabb korlátokat eredményez.

A borotválás egy lépését az alábbi programkóddal végezhetjük el:

```
% 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).
```

7.2.4. További finomhangolási lehetőségek

A szomszédsági reláció megvalósítására több lehetőség is kínálkozik:

- A vízszintes és függőleges szomszédsági reláció egyaránt megvalósítható a `relation/3` meghívásával vagy indexikálisként való fordításával is. Ezek között az opciólistában a `relation(R)` elemmel kapcsolgathatunk (`R = clause` vagy `R = indexical`). Alapértelmezésként a korlát indexikálisként van megvalósítva.

- Az átlós szomszédsági relációt is többféleképpen megvalósíthatjuk. A kívánt variánst a `diag(D)` opcióval választhatjuk ki, ahol `D` lehet:

  - `reif` — reifikációs alapon: `CF1 #= 0 #\/ CF2 #= 0`
  - `ind_arith` — aritmetikát használó indexikálissal:
    `diagonal_neighbour_arith(CF1, CF2) +:`
    `CF1 in 0 .. (1000-(min(CF2)/>1000)*1000), ...`
  - `ind_cond` (alapértelmezés) — feltételes indexikálissal: `diagonal_neighbour_cond(CF1, CF2) +:`
    `CF1 in (min(CF2)..0) ? (inf..sup) \/ 0, ...`

7.2.5. Futási eredmények

Az alábbi időeredmények az összes megoldás megtalálására vonatkoznak, és egy DEC Alpha 433 MHz-es gépen születtek. A táblázatokban lévő adatpárok a futási időt (mp) és a visszalépések számát jelentik.

Opciók/példa	`fules2a`		`fules3`		`fules_clean`	
1. sima	51.437	10178	253.1	55157	1085.7	260K
Redundáns korlátok						
2. = 1 + `count_ships_occs`	16.218	1910	105.6	13209	395.2	52398
3. = 2 + `count_ones_columns`	16.175	1861	105.0	12797	386.4	50181
4. = 3 + `count_empties`	17.915	1771	107.2	11273	381.7	42417
Címkézési variánsok						
5. = 4 + `label(max_dual)`	18.296	1771	106.3	11273	379.8	42417
6. = 4 + `label(ships)`	17.153	1708	105.7	11236	367.8	41891
Borotválás						
7. = 6 + `filter([repetitive])`	10.517	313	64.3	2534	206.1	10740
8. = 6 + `filter([on])`	9.549	332	59.0	2811	199.7	12004
Megvalósítási variánsok						
9. = 8 + `relation(indexical)`	8.426	332	54.0	2811	180.8	12004
10.= 9 + `diag(ind_arith)`	7.855	332	50.2	2811	167.7	12004
11.= 9 + `diag(ind_cond)`	*7.819*	*332*	*50.1*	*2811*	*166.2*	*12004*
12.= 11 − `count_empties`	6.750	350	47.5	3248	166.2	14233

Jelmagyarázat:

1. sima = `[-count_ships_occs,-count_ones_columns,-count_empties, label(plain),filter([off]),relation(clause),diag(reif)]`
11. = alapértelmezés

7.3. Dominó

Adott egy $(n+1)\times(n+2)$-es téglalap, amelyen egy teljes n-es dominókészlet összes elemét elhelyeztük, majd a határokat eltávolítottuk. A feladat a határok helyreállítása a számok alapján. A dominókészlet elemei az $\{\langle i,j\rangle \,|0 \leq i \leq j \leq n\}$ számpároknak felelnek meg. A kiinduló adat tehát egy $0..n$ intervallumbeli számokból álló $(n+1)\times(n+2)$-es mátrix, amelynek elemei azt mutatják meg, hogy az adott mezőn hány pöttyöt tartalmazó féldominó van.

Az alábbi ábrán látható egy feladat $n = 3$-ra, és annak egyetlen megoldása:

```
                                 ---------------------
    1   3   0   1   2            | 1 | 3   0 | 1 | 2 |
                                 |   |-------|   |   |
    3   2   0   1   3            | 3 | 2   0 | 1 | 3 |
                                 |---------------|---|
    3   3   0   0   1            | 3   3 | 0   0 | 1 |
                                 |-------|-------|   |
    2   2   1   2   0            | 2   2 | 1   2 | 0 |
                                 ---------------------

% Bemenő adatformátum:           % A megoldás Prolog alakja:

[[1,  3,  0,  1,  2],            [[n,  w,  e,  n,  n],
 [3,  2,  0,  1,  3],             [s,  w,  e,  s,  s],
 [3,  3,  0,  0,  1],             [w,  e,  w,  e,  n],
 [2,  2,  1,  2,  0]]             [w,  e,  w,  e,  s]]
```

A megoldásban a téglalap minden mezőjéről el kell dönteni, hogy azon a mezőn egy dominó északi (`n`), déli (`s`), keleti (`e`) vagy nyugati (`w`) fele van. A `http://www.cs.bme.hu/~szeredi/oktatas/nlp/hf_00s_domino.tgz` címről letölthető állomány tesztadatai négy csoportba oszthatóak:

- `base` — 16 könnyű alapfeladat n = 1–25 közötti méretben.
- `easy` — 24 középnehéz feladat, többségük n = 15–25 méretben.
- `diff` — 21 nehéz feladat 28-as, és egy 30-as méretben.
- `hard` — egy nagyon nehéz feladat 28-as méretben.

7.3.1. A feladat modellezése

A torpedó feladathoz hasonlóan itt is először érdemes meggondolnunk, hogy a modellezéshez milyen változókat használjunk, és ezeknek milyen értékkészletet válasszunk. A korlátváltozók bevezetésére fennálló lehetőségek:

a. Minden mezőhöz egy ún. *irányváltozó*t rendelünk, amely a lefedő féldominó irányát jelzi (ez az, ami a megoldásban is szerepel). Ezzel a megoldással az a baj, hogy körülményes a dominók egyszeri felhasználását biztosítani.

b. Minden dominóhoz egy ún. *dominóváltozó*t rendelünk, amelynek értéke megmondja, hová kerül az adott dominó. Ezzel a megoldással viszont az a probléma, hogy körülményes a dominók át nem fedését biztosítani.

c. Mezőkhöz és dominókhoz is rendelünk változókat (a.+b.). Ez az *egyik* választott megoldás.

d. A mezők közötti választóvonalakhoz rendelünk egy 0-1 értékű ún. *határváltozó*t, amely azt mutatja meg, hogy az adott választóvonalon egy dominó közepe van-e, vagy pedig két dominó érintkezik rajta. Ez a *másik* választott megoldás.

Az irányváltozók értékkészletét legegyszerűbb az `n`, `s`, `e`, `w` konstansok valamilyen numerikus kódolása szerint megválasztani. A dominóváltozók értékkészletét lehetne egy $\langle$*sor,oszlop,lehelyezési_irány*$\rangle$ hármassal modellezni, de egyszerűbb megszámozni egy adott dominó l lehetséges lehelyezési módját, és az 1..l számozást használni. Például a fenti elrendezésben a 0/2-es dominó csak három különböző módon rakható le: $\langle$2, 2, vízsz$\rangle$, $\langle$3, 4, függ$\rangle$ és $\langle$4, 4, vízsz$\rangle$. Így a dominónak megfeleltetett változót az 1..3 értéktartományra szoríthatjuk be. A határváltozók 1 értékének „természetes" jelentése lehetne, hogy az adott határvonalat az ábrán be kell húzni. Érdemes azonban ennek negáltjával dolgozni: legyen 1 az érték akkor, ha az adott vonal egy dominó középvonala. Ez azért jó, mert ettől az összes korlát `A+B+... #= 1` alakú lesz.

7.3.2. Egy lehetséges megoldás

Változók, korlátok

- Minden mezőhöz egy irányváltozó (`I`yx `in 1..4` $\equiv$ `{`n,w,s,e`}`), minden dominóhoz egy dominóváltozó (`D`$ij, 0 \le i \le j \le n$) tartozik.

- Szomszédsági korlát: két szomszédos irányváltozó kapcsolatát adja meg, pl. `I14#=n #<=> I24#=s`, `I14#=w #<=> I15#=e`, stb. Egyszerűen az olyan jellegű feltételeket adja meg, mint pl. „ha egy irányváltozó értékét mondjuk `n`-nek (dominó északi fele) választjuk, akkor az alatta lévő mező csak `s` (dominó déli fele) lehet".
- Dominó-korlát: egy dominó-elhelyezésben a dominóváltozó és a lerakás bal vagy felső mezőjének irányváltozója közötti kapcsolat. A korábbi példában pl. `D02#=1 #<=> I22#=w`, `D02#=2 #<=> I34#=n`, `D02#=3 #<=> I44#=w`.

Mivel mindegyik megvalósítandó korlát az „akkor és csak akkor" logikai kapcsolatra épül, ezért az egyetlen fontos feladatunk, hogy ezt a logikai kapcsolatot hogyan lehet az optimálisat megvalósítani. A mintamegoldás három verziót valósít meg a `csakkor_egyenlo(X,C,Y,D)` $\equiv$ `X #= C #<=> Y #= D` korlátra:

- `reif`: reifikációval (`X#=C#<=>Y#=D`)
- `ind1`: az `'x=c=>y=d'` FD predikátum kétszeri hívásával,
- `ind2`: az `'x=c<=>y=d'` FD predikátum hívásával.

Ezek közül az opciólista `csakkor=Cs` paraméterével választhatunk, ahol `Cs` helyére kell írni a megfelelő variáns nevét. Az `ind1`, illetve `ind2` variánshoz használt FD predikátumok:

```
'x=c=>y=d'(X, C, Y, D) +:
        X in (dom(Y) /\ {D}) ? (inf..sup) \/ \({C}),
        Y in ({X} /\  \({C})) ? (inf..sup) \/ {D}.

'x=c<=>y=d'(X, C, Y, D) +:
        X in ((dom(Y) /\ {D}) ? (inf..sup) \/ \({C})) /\
             ((dom(Y) /\ \({D})) ? (inf..sup) \/ {C}),
        Y in ((dom(X) /\ {C}) ? (inf..sup) \/ \({D})) /\
             ((dom(X) /\ \({C})) ? (inf..sup) \/ {D}).
```

A címkézésnél két, lényegében különböző lehetőségünk van: címkézhetünk az irányváltozók és a dominóváltozók szerint. Ezeken belül még variálhatunk a `labeling/2` paraméterezésével is. Az opciólistában a `valt=V` opció szolgál az irányváltozók és a dominóváltozók közti váltásra (`V=irany` az irányváltozók címkézése, `V=domino` a dominóváltozók címkézése), a `labeling/2` járulékos paramétereit a `label=LOpciok` segítségével adhatjuk át. A borotválás finomhangolása a `szur=Sz` és `szurtek=L` opciók alkalmazásával végezhető el. Ha `szur` $\neq$ `ki`, akkor az irány-változókat borotváljuk, sorra megpróbáljuk az `L` elemeire behelyettesíteni, és ha ez meghiúsulást okoz, akkor az adott elemet kivesszük a változó tartományából. `szur` lehet: `elott` (csak a címkézés előtt szűrünk) vagy `N` (minden `N`. változó címkézése után szűrünk). `L` alapértelmezése `[w,n]`.

7.3.3. Egy másik lehetséges megoldás

Változók, korlátok

- Minden mező keleti ill. déli határvonalához egy-egy határváltozó tartozik (Eyx ill. , Syx). A határváltozó akkor és csak akkor 1, ha az adott vonal egy dominó középvonala. A táblázat külső határai 0 értékűek (behúzott vonalak).

- Szomszédsági korlát: minden mező négy oldala közül pontosan egy lesz egy dominó középvonala, tehát pl. a $(2,4)$ koordinátájú dominó-mező esetén `sum([S14,E23,S24,E24]), #=, 1)`.

- Lerakási korlát: egy dominó összes lerakási lehetőségeit tekintjük, ezek középvonalai közül pontosan egy lesz 1, így a példabeli $\langle 0, 2\rangle$ dominóra: `sum([E22,S34,E44], #=, 1)`.

Az előző változathoz hasonlóan itt is lényegében egyetlen korlát minél optimálisabb megvalósításával kell foglalkozni. Ez a korlát egy változólistát kap paraméterül, és azt a feltételt fejezi ki, hogy a lista összegének 1-nek kell lennie (`lista_osszege_1`). A lehetséges megvalósítások (az `osszeg=Ossz` opción keresztül választhatunk közöttük):

- `Ossz=ari(N)`: `N`-nél nem hosszabb listákra aritmetikai korláttal, egyébként (`N`-nél hosszabb listákra) a `sum/3` korláttal

- `Ossz=ind(N)`: `N`-nél nem hosszabb listákra FD predikátummal, egyébként (`N`-nél hosszabb listákra) a `sum/3` korláttal

- `Ossz=sum`: mindig a `sum/3` korláttal

Mivel az FD predikátumok kötött számú változóval dolgoznak, ezért az `Ossz=ind(N)` esetben szét kell választanunk az egyes eseteket, valahogy így:

```
osszege1(A, B) +:             A+B #= 1.
osszege1(A, B, C) +:          A+B+C #= 1.
osszege1(A, B, C, D) +:       A+B+C+D #= 1.
(...)
```

7.3.4. Futási eredmények

Az alábbi időeredmények az összes megoldás megtalálására vonatkoznak, és egy DEC Alpha 433 MHz-es gépen születtek. A táblázatokban lévő adatpárok a futási időt (mp) és a visszalépések számát jelentik. A dőlt betűs sorok a viszonyítási alapot jelzik, a felkiáltójel azt mutatja, hogy időtúllépés is volt a tesztadatok között. A keretezés a legjobb időt, illetve visszalépés-számot jelenti.

Opciók/példa	base		easy		diff		hard	
1. változat,csakkor=ind1,valt=domino,label=[],szur=2,szurtek=[1,2]								
szur=2	*5.44*	*1*	*26.6*	*28*	*4001.7*	*4950*	*1162.9*	*1448*
szur=1,label=[ff]	5.87	1	27.6	5	3900.6	1168	554.4	$\boxed{159}$
szur=2,label=[ff]	5.48	1	25.8	13	3222.9	2074	446.9	288
szur=3,label=[ff]	5.36	1	25.7	19	3232.6	3597	$\boxed{429.3}$	477
label=[ffc]	5.49	1	23.7	7	!9885.8	6403	3902.0	2795
csakkor=ind2	5.14	1	26.4	28	4250.9	4950	1233.0	1448
csakkor=reif	6.87	1	33.5	28	4573.2	4950	1320.2	1448
szurtek=[1]	4.98	9	34.1	92	6375.0	13824	1976.5	3566
szur=elott	5.09	1	25.1	1722				
szur=ki	38.6	9K	590	157K				
1. változat,csakkor=ind1,valt=irany,label=[],szur=2,szurtek=[1,2]								
label=[]	*5.39*	*1*	*23.4*	*10*	*2138.1*	*1377*	*3362.9*	*2326*
label=[ff]	5.40	1	23.4	10	2137.9	1377	3376.5	2326
label=[ffc]	5.42	1	24.1	10	!15036.1	10155	!7199.7	4380
szurtek=[1]	4.94	3	29.4	45	3240.2	4000	6077.2	7782
2. változat,osszeg=ind(5),label=[],szur=2,szurtek=[1]								
szur=2	*2.10*	*1*	$\boxed{11.5}$	*8*	$\boxed{1045.9}$	*1399*	*1607.0*	*2254*
szur=1	2.28	1	11.9	$\boxed{3}$	1294.7	$\boxed{787}$	1977.9	1277
szur=3	2.04	1	11.5	20	1051.2	2436	1583.1	3851
osszeg=ind(4)	2.18	1	11.9	8	1152.7	1399	1768.0	2254
osszeg=ind(6)	2.13	1	11.9	8	1149.2	1399	1765.5	2254
osszeg=sum	2.96	1	15.8	8	1409.3	1399	2263.1	2254
osszeg=ari(5)	2.97	1	15.9	8	1462.7	1399	2257.8	2254
szurtek=[0]	$\boxed{1.86}$	2	15.1	103	2104.6	10719	3211.3	17300
szurtek=[0,1]	2.00	1	12.3	7	1182.2	1324	1823.7	2150
label=[ff]	2.12	1	11.7	8	1132.3	1399	1735.2	2254
label=[ffc]	2.14	1	12.4	8	2189.5	2841	2672.1	3732
2. változat,szur=ki,label=[], rövidítések: l => lerak sz => szomsz								
osszeg=ind(5)	3.31	818	57.0	21181				
l=ind(5),sz=sum	4.61	818	78.6	21181				
l=sum,sz=ind(5)	3.97	818	62.8	21181				
osszeg=sum	4.57	818	74.8	21181				

8. fejezet

A Mercury nagyhatékonyságú LP megvalósítás*

Ebben a fejezetben a Mercury nagyhatékonyságú logikai programnyelvet mutatjuk be nagy vonalakban. Mivel a Prolog önmagában nem kimondottan alkalmas nagyobb méretű projektek kezelésére, a Mercury megalkotásakor a készítők célja elsődlegesen a nagybani programozás támogatása, valamint a produktivitás, a megbízhatóság és a hatékonyság növelése volt. Eközben a következő irányelveket tartották szem előtt:

- 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

A Mercury nyelv és az implementáció fejlesztője a University Of Melbourne. A nyelv honlapja a `http://www.cs.mu.oz.au/mercury/` címen található meg.

8.1. Egy Mercury példaprogram

A feladat: operációs rendszerek file-név-illesztéséhez hasonló funkció megvalósítása. Egy karaktersorozat illesztésekor a `?` karakter egy tetszőleges másik karakterrel illeszthető, a `*` karakter egy tetszőleges (esetleg üres) karaktersorozattal illeszthető, a `\`c karakter-pár pedig a c karakterrel illeszthető. Ha egy minta `\`-re végződik, az illesztés meghiúsul. Bármely más karakter csak önmagával illeszthető.

A program hívási formája: `match Pattern1 Name Pattern2`, ahol a `Pattern1` és `Pattern2` mintákban a `*` és `?` karaktereknek azonos elrendezésben kell előfordulniuk.

A program funkciója: a `Pattern1` mintára az összes lehetséges módon illeszti a `Name` nevet, a `*` és `?` karakterek helyére kerülő szöveget `Pattern2`-be behelyettesíti, és az így kapott neveket kiírja.

```
:- 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")
    ).

:- 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).
```

A program egy futása

```
# ./match **z?c  foozkc '|*|*|?'
Pattern '**z?c' matches 'foozkc' as '|*|*|?' matches the following:

    |foo||k
    |fo|o|k
    |f|oo|k
    ||foo|k

*** No (more) solutions
```

8.2. A Mercury modul-rendszere

A Mercury programokat különálló modulokból lehet felépíteni, ez lehetővé teszi a modulok egymástól független, szeparált fordítását. A modul-rendszer támogatja az absztrakt típusok használatát és a modulok egymásba ágyazhatóságát is.

Egy Mercury modult mindig a `:- module` ⟨modulename⟩`.` sorral kezdjük (a fenti példában `:- module match.`). A modul két részre osztható, az *interfész* és az *implementációs* részre. Az interfész részt az `:- interface.` kulcsszóval kezdjük. Ebben a részben minden szerepelhet, kivéve függvények, predikátumok és almodulok definícióját. Az itt lévő dolgok fognak „kilátszani" a modulból. Az implementációs részt az `:- implementation.` kulcsszó vezeti be, és ide kerülnek az interfész részben deklarált függvények, predikátumok, absztrakt adattípusok és almodulok pontos definíciói. Az implementációs rész tartalma lokális a modulra nézve.

Egy modul természetesen felhasználhatja egy másik modul interfész részét. Ehhez a használni kívánt modult importálni kell az `:- import_module` ⟨modules⟩`.` vagy az `:- use_module` ⟨modules⟩`.` predikátum használatával. A kettő között az a lényegi különbség, hogy az `import_module` alkalmazásával importált modulokból származó predikátumok használatához nincs szükség külön modul-kvalifikációra, míg a `use_module` esetében igen. A modul-kvalifikáció alakja: ⟨module⟩:⟨submodule⟩:…:⟨submodule⟩:⟨name⟩. A Mercury jelenlegi verzióiban a `:` helyett egyelőre a `__` (dupla aláhúzásjel) javasolt, mert lehet, hogy a későbbiekben a `:` helyett a `.` lesz a modulkvalifikátor és a `:` a típuskvalifikátor. Modulkvalifikációra egyébként a mintaillesztő program interfész részében a `:- pred main(io__state::di, io__state::uo) is det.` sorban láthatunk egy egyszerű példát.

Almodulok használata elvileg kétféleképpen is lehetséges: a főmodul fájljába beágyazva (*beágyazott almodul*) vagy külön fájlban (*szeparált almodul*). A jelenlegi implementációban a beágyazott almodulok használata egyelőre problémás.

8.3. A Mercury típusrendszere

A Mercury a Prologtól eltérően egy típusos nyelv. A típusoknak több fajtája lehet:

- 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

A felhasználói adattípusok között szerepel az SML-ből ismerős *megkülönböztetett unió* adattípus (SML-ben ez `datatype` néven szerepelt), az *ekvivalencia típus* (más néven típusátnevezés, SML-ben `type` néven szerepelt) és az *absztrakt típus* is. A megkülönböztetett uniónak is több alfajtája van:

- *Enumeráció típus*
  `:- type fruit ---> apple; orange; banana; pear.`
  A fenti példában egy `fruit` nevű típust definiálunk, amely változói négy lehetséges értéket vehetnek fel, és ezeket az értékeket rendre az `apple`, `orange`, `banana`, `pear` névkonstansokkal azonosítjuk.
- *Rekord típus*
  `:- type itree ---> empty; leaf(int); branch(itree, itree)`
  Ez a példa egy `itree` nevű típust definiál, amely bináris fák reprezentálására szolgál. Egy `itree` típusú változó értéke háromféle lehet: `empty` (ez jelképezi az üres fát), `leaf(int)` (ez jelképez egy levelet, a levél értékének típusa `int`) és `branch(itree, itree)` (ez egy elágazó csomópontot jelképez, ahol mindkét ág egy-egy újabb `itree` típusú objektum).
- *Polimorfikus típus*

  ```
  :- type list(T) ---> [] ; [T|list(T)].
  :- type pair(T1, T2) ---> T1 - T2.
  ```

  A `list(T)` típus egy `T` típusú elemekből álló lista, ahol `T` maga egy *típusváltozó*. Így például a `list(int)` `int` típusú elemek listáját adja, `list(char)` karakterlistát, és így tovább. A `pair(T1, T2)` típus `T1` és `T2` típusú elemekből álló párok típusa, ahol `T1` és `T2` szintén típusváltozók. Például `pair(int,char)` egy olyan párt ír le, amely első tagja `int`, második tagja `char` típusú.

A megkülönböztetett unió leírásának megalkotására vonatkozó általános szabályok:

- A típusleírás alakja: `:- 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

Az ekvivalencia típus leírása:

- `:- type` ⟨típus⟩ `==` ⟨típus⟩`.` (például `:- type assoc_list(K, V) == list(pair(K, V)).`)

- nem lehet ciklikus
- a jobb és a bal oldal teljesen ekvivalens egymással

Az absztrakt típusra az a jellemző, hogy az interfész részben csak a típus neve van megadva (`:- type` ⟨típus⟩`.`), a típus tényleges definíciója az implementációs részben el van rejtve.

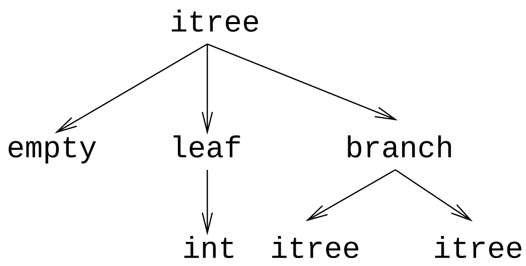
A típusoknak a predikátum- és függvényleírásoknál van szerepe: egy predikátum vagy függvény leírásánál mindig meg kell adni a paraméterek típusát. Például:

```
:- pred is_all_uppercase(string).
:- func length(list(T)) = int.
```

8.4. Módok és behelyettesítettség

8.4.1. definíció: egy paraméter *mód*jának nevezünk egy két behelyettesítettségi állapotból álló párt, ahol az első állapot azt jelöli, ahogyan a paraméter bemegy egy adott predikátumba, a második pedig azt, ahogy kijön belőle. Például az `out` mód jelentése: szabad változó megy be, tömör kifejezés jön ki.

Saját adattípusainkhoz különböző, részleges behelyettesítettséget leíró behelyettesítettségi állapotot is rendelhetünk. Tekintsük például a bináris fa adattípusát:



Az állapot leírásakor a típust tartalmazó („vagy”) csúcsokhoz rendelünk behelyettesítettségi állapotot. A deklarációban a `bound/1`, a `free/0` és a `ground/0` funktorokat használhatjuk. Például:

```
:- inst bs = bound(empty; leaf(free); branch(bs,bs)).
```

A fenti deklaráció alapján `bs` behelyettesítettségű az olyan bináris fa, amely vagy üres (`empty`), vagy egy szabad változót tartalmazó levél (`leaf(free)`), vagy egy olyan elágazás, amely mindkét fele `bs` behelyettesítettségű. Lehetőség van parametrizált `inst`-ek készítésére is:

```
:- inst bs(Inst) = bound(empty ; leaf(Inst) ; branch(bs(Inst),bs(Inst))).
:- inst listskel(Inst) = bound([] ; [Inst|listskel(Inst)]).
```

Egy mód leírása a behelyettesítettségi állapotok alapján a következőképpen néz ki: `:- mode` ⟨m⟩ `==` ⟨inst1⟩ `>>` ⟨inst2⟩`.`, ahol ⟨m⟩ a mód neve, ⟨inst1⟩ és ⟨inst2⟩ pedig behelyettesítettségi állapotok nevei. Az `in` és az `out` mód leírása például így néz ki:

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

Lehetőség van módok átnevezésére is: `:- mode` ⟨m1⟩ `==` ⟨m2⟩.

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

Természetesen a parametrizálás lehetősége a móddeklarációknál is adott:

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

A módokat a predikátum-mód deklarációkban lehet hasznosítani. Itt egy predikátumról azt írjuk le, hogy milyen mód-kombinációkban szabad használni a predikátum paramétereit. Például a `lists` könyvtár `append/3` eljárásának mindhárom bemeneti paramétere `list(T)` típusú (ahol `T` tetszőleges típusnév), és két módban használható. Az egyik módban az első két paraméter bemeneti, a harmadik kimeneti, és így az eljárás listák összefűzésére szolgál. A másik módban az első két paraméter kimeneti, a harmadik bemeneti, és így az eljárás a harmadik listát az összes lehetséges módon szétszedi. Ezt predikátum-mód deklarációval a következőképpen lehet leírni:

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

Ha csak egyetlen mód van, akkor azt össze lehet vonni a `pred` deklarációval:

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

Amire még figyelni kell a predikátum-mód deklarációkkal kapcsolatban:

- `free` változókat még egymással sem lehet összekapcsolni, ezért az alábbi példa hibás:

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

- 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! Lásd a `match_list/3` `append/3` hívását!) Az átrendezés nem okozhat problémát, mivel a Mercury tisztán logikai nyelv, ezért a hívások sorrendje tetszőleges.
- A megadottnál „jobban” behelyettesített argumentumokat egyesítésekkel kiküszöböli a fordító. Ezeket a módokat le sem kell írni (de érdemes lehet). Például `:- mode append(in, out, in).` a szétszedő `append`-et fogja használni.
- A jelenlegi implementáció nem kezeli a részlegesen behelyettesített adatokat.

8.5. Determinizmus

A Mercury-ban lehetőség van arra is, hogy minden predikátum minden módjára (azaz minden eljárásra) megadjuk, hogy hányféleképpen sikerülhet, és hogy meghiúsulhat-e. Hatféle determinizmus kategória létezik:

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

Egy determinizmus-deklaráció formailag majdnem megegyezik egy egyszerű predikátum-mód deklarációval, a különbség mindössze annyi, hogy a deklaráció végén egy `is` kulcsszó után oda kell írni a megfelelő determinizmus kategóriát is. Például az `append/3`-ra:

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

Ha csak egyetlen mód van, akkor azt össze lehet vonni a `pred` deklarációval:

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

Felvetődik a kérdés, hogy mi értelme van a két „egzotikus” determinizmusnak, a `failure`-nak és az `erroneous`-nak. A `failure` egy olyan predikátumot jelent, amely soha nem ad megoldást, és mindig meghiúsul. Ilyen például a `fail/0` eljárás. `erroneous` determinizmussal a `require_error/1` eljárás rendelkezik, amely hibaüzenetek kijelzésére alkalmas (az `erroneous` determinizmus elvileg egy olyan eljárást jelent, amely soha nem ad megoldást, de nem is hiúsul meg). Ez a két determinizmus például hibakezelésre alkalmazható:

```
:- mode append(out, out, out) is erroneous.
```

Ehhez természetesen írni kell egy olyan klózt is, amely mindhárom paraméter behelyettesítetlensége esetén a `require_error/1` használatával hibát jelez.

Függvények esetén ha minden argumentum bemenő, akkor a determinizmusuk csak `det`, `semidet`, `erroneous` vagy `failure` lehet, a többszörös megoldásokat nem szabad megengednünk, mert ilyen esetben nem is beszélhetünk függvényről. Például a `between(in, in, out)` nem írható le függvény alakban.

8.6. Magasabbrendű eljárások

A hagyományos Prolog megvalósításban a `call/1` eljárás segítségével lehetőség volt arra, hogy egy eljárásból egy másik eljárást hívjunk meg. A Mercury-ban bevezették a `call/2`, `call/3` …eljárásokat is, amelyek segítségével úgy hívhatunk meg egy eljárást, hogy annak paraméterlistáját a `call`-ban átadott paraméterekkel kiegészítjük. Például a `call/4` definíciója Prologban:

```
% 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).
```

Az ilyen jellegű `call` hívások segítségével lehetőség nyílik magasabb rendű eljárások egyszerű megvalósítására. Például az SML-ből ismert `map` funkció:

```
% 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(_, [], []).
```

Itt `Pred` egy olyan eljárás, amelynek két paramétere van, és amely a két paraméter között egy transzformációt valósít meg. A `map` eljárás módjai és determinizmusa a `Pred` eljárástól függ, hiszen ha `Pred` mondjuk `pred(in, out) is multi`, akkor a `map` második és harmadik paramétere is rendre `in` és `out` lesz, `map` determinizmusát pedig `Pred` determinizmusa fogja megszabni. Néhány (szám szerint 5) lehetőség leírása a fenti programkódban is megtalálható. A `map` használata:

```
:- 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).
```

Itt `negyzet/2` egy olyan predikátum, amely a négyzetre emelés transzformációját valósítja meg. Érdemes észrevenni, hogy a Mercury-ban kikerülhetjük az aritmetikai számításoknál az `is`-zel való vacakolást, hiszen mivel a `negyzet/2` predikátum-mód deklarációja tartalmazza, hogy a második paraméter is `int`, ebből a Mercury már rájön, hogy az `X*X` kifejezés nem egy `'*'/2` struktúrát jelent, hanem egy szorzást.

Az SML-ből ismert λ (*lambda*) kifejezés Mercury-ban is használható, így a fenti két predikátumot egyetlen predikátumba vonhatjuk össze:

```
:- 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).
```

Itt a négyzetre emelés predikátumát a `map` hívásba ágyazottan egy névtelen λ-függvény segítségével valósítjuk meg.

Magasabbrendű kifejezéseket háromféleképpen hozhatunk létre:

- 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 a fenti példákban `pred(list(int), int)`.

Magasabbrendű függvények létrehozására is háromféle lehetőség van:

- 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
  ```

Az SML-ből ismert „curry”-zés mechanizmusa Mercury-ban is működik (kivéve a beépített nyelvi konstruktorokra (pl. `=`, `\=`, `call`, `apply`), ezeket nem lehet „curry”-zni:

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

A DCG nyelvtanok leírásánál külön szintaxis van az olyan eljárásokra, amelyek egy akkumulátor-párt is használnak:

```
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
)
```

Magasabbrendű eljárásokat kétféleképpen hívhatunk meg:

- `call(Closure, Arg`$_1$`, …, Arg`$_n$`)` ($n \geq 0$)— a `Closure` argumentumlistája kiegészül az `Arg`$_1$`, …, Arg`$_n$ argumentumokkal, és úgy hívódik meg.
- `solutions(match(P1, N1, P2), Sols)` — összegyűjti a `match(P1, N1, P2)` hívás összes megoldását `Sols`-ba. Természetesen `match(P1, N1, P2)` helyett tetszőleges más predikátum is használható.

Függvények meghívására az `apply` használható: `apply(Closure2, Arg`$_1$`, …, Arg`$_n$`)` ($n \geq 0$) hívásakor a `Closure2` argumentumlistája kiegészül az `Arg`$_1$`, …, Arg`$_n$ argumentumokkal, és úgy hívódik meg, `apply` eredménye a hívás eredménye lesz. Használata: `List = apply(Double, [1,2,3])`.

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.

8.7. Problémák a determinizmussal

A determinizmus bevezetésével ésszerű néhány korlátozást bevezetni. Ilyen például az, hogy `det` vagy `semidet` módú eljárásokból nem hívhatunk `nondet` vagy `multi` eljárást, hiszen ezzel elrontjuk a determinizmust. A Mercury programoknál a főprogramnak (`main/2`) szükségszerűen `det`-nek kell lennie, ezzel viszont látszólag elvesztettük a `nondet` és a `multi` eljárások hívásának lehetőségét, hiszen a főprogram `det`, és belőle nem hívhatóak `nondet` és `multi` eljárások. Ilyen eljárások hívásakor döntenünk kell:

- Ha az összes megoldást akarjuk, akkor a `std_util__solutions/2` használatával explicit módon meg kell kerestetnünk az összes megoldást. Ezzel a `nondet` vagy `multi` eljárást `det`-té tudjuk tenni, mert a `std_util__solutions/2` mindig sikerülni fog
- Ha csak egy megoldás akarunk, és mindegy, hogy melyiket, akkor vagy azt csináljuk, hogy az eljárás kimenő változóit nem használjuk fel (mert ekkor az első utáni megoldásokat levágja a rendszer), vagy pedig az úgynevezett *committed choice nondeterminism* kihasználásával (`cc_nondet`, `cc_multi` determinizmus) determinizáljuk a nemdeterminisztikus eljárásokat. A *committed choice nondeterminism* mechanizmust olyan helyeken használjuk, ahol biztosan nem lesz szükség több megoldásra. Fontos megjegyezni, hogy IO műveletek csak `det`, `cc_nondet` és `cc_multi` eljárásokban használhatóak, mivel az IO műveleteket a visszalépés során nem lehet visszavonni, ezért fontos, hogy csak olyan helyeken lehessen őket alkalmazni, ahol biztosan nem lesz visszalépés.
- Ha csak néhány megoldást akarunk, akkor a `std_util__do_while/4` eljárást használhatjuk.

Arra az esetre, ha egy olyan eljárást akarunk meghívni, amelynek minden megoldása ekvivalens, még nincs igazi megoldás. A tervekben a `unique [X] goal(X)` szerkezet szerepel, de egyelőre még a C interfésszel kell trükközni ilyen esetekben.

Tekintsük például az alábbi feladatot: soroljuk fel egy halmaz összes részhalmazát, és minden megoldást pontosan egyszer adjunk ki! Egy halmaz egy részhalmazának kiválasztása nyilvánvalóan többféleképpen is sikerülhet, ezért valamilyen módon (például a `cc_multi` determinizmus használatával) ki kell küszöbölni a nemdeterminizmust:

```
:- 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("Set version:\n"),
        {std_util__unsorted_solutions(resze(S), P)},
        io__write_list(P, " ", io__write),
        io__write_string("\n\nList 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(L0) -> L = L0, set__list_to_set(L, S)
        ;   set__init(S), L = []
        }.
```

A `resze(S)` a halmazokat `set` absztrakt adattípussal fogja ábrázolni, az `lresze(L)` pedig `list` adattípussal. A kétféle megoldás:

```
:- 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
        ).

:- 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
        ).
```

A `set__member/2` felsoroló jellege miatt nem teljesíti azt a feltételt, hogy minden megoldás pontosan egyszer jelenjen meg, a lista fejének leválasztása viszont (szemi)determinisztikus, ezért a `set`-ekkel operáló verzió többször is kiad egy megoldást, míg a listákkal operáló nem:

```
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].
Set version:
[1, 2] [2] [1] [] [1, 2] [1] [2] []

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

Egy másik `cc_multi` példa az N királynő feladat megoldására (egyszerű generate-and-test jellegű megoldás, ahol `perm/2` generálja a királynők elrendezéseit, `safe/1` pedig ellenőriz):

```
:- 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).
```

9. fejezet

CHR—Constraint Handling Rules*

A CHR (*Constraint Handling Rules*) egy deklaratív nyelv-kiterjesztés, amely determinisztikus kifejezés-átíráson alapul. Prolog, CLP, Haskell vagy Java *gazda*nyelvbe ágyazódva képes működni. Általános szimbolikus (nem numerikus) felhasználói korlátok írására alkalmas, de nem tartalmaz konzisztencia-vizsgálatot, erről a megfelelő szabályok megírásával nekünk kell gondoskodnunk. Fő szerzője Thom Frühwirth (ECRC, LMU München, Ulm Uni.).

A nyelv-kiterjesztés honlapja: `http://www.pst.informatik.uni-muenchen.de/~fruehwir/chr-intro.html`.

SICStus Prologban a CHR kiterjesztést a következő paranccsal vehetjük használatba:

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

9.1. CHR szabályok

A CHR kiterjesztésben a korlát-tár alakulását saját magunk által írt szabályok segítségével írhatjuk le. A szabályoknak három fajtája van:

- *Egyszerűsítés* (*simplification*):
  H_1, ..., H_i `<=>` G_1, ..., G_j `|` B_1, ..., B_k.
- *Propagáció* (*propagation*):
  H_1, ..., H_i `==>` G_1, ..., G_j `|` B_1, ..., B_k.
- *Egypagáció* (*simpagation*):
  H_1, ..., H_l \ H_{l+1}, ..., H_i `==>` G_1, ..., G_j `|` B_1, ..., B_k.

Amint látható, egy CHR szabály alapvetően három részre osztható:

- *multi-fej* (*multi-head*): ez alatt a „H_1, ..., H_i” részt értjük, ahol a H_m-ek CHR-korlátok
- *őr* (*guard*): G_1, ..., G_j, ahol a G_m-ek a gazda-nyelvben értelmezett korlátok
- *törzs* (*body*): B_1, ..., B_k, ahol a B_m-ek CHR- vagy gazda-korlátok
- mindvégig $i > 0, j \geq 0, k \geq 0, l > 0$.

A fentiekben mindvégig $i > 0, j \geq 0, k \geq 0, l > 0$.

Az *egyszerűsítés* azt fejezi ki, hogy ha az őrben felírt korlátok igazak, akkor a fej ekvivalens a törzzsel, ezért a fejet ki lehet törölni a korlát-tárból, és helyette a törzset fel lehet venni. A *propagáció* esetében ha az őr igaz, akkor a fejből következik a törzs, ezért a törzset fel lehet venni a korlát-tárba. Az *egypagáció* az egyszerűsítés és a propagáció összegyúrásából keletkezik, és vissza is lehet vezetni rájuk, hiszen: `Heads1 \ Heads2 <=> Body` ugyanazt jelenti, mint `Heads1, Heads2 <=> Heads1, Body`, csak sokkal hatékonyabb, mert `Heads1`-et nem kell újra felvenni a korlát-tárba.

9.2. A CHR szabályok végrehajtása

A CHR korlátoknak három állapota lehet: *aktív*, *aktiválható passzív* és *alvó passzív*. Aktív korlátból legfeljebb egy van minden pillanatban. Egy korlát akkor válik aktiválhatóvá, ha valamelyik változóját *megérintik*, azaz egyesítik egy tőle különböző kifejezéssel. Minden alkalommal, amikor egy korlát aktívvá válik, az összes rá vonatkozó szabályt végigpróbáljuk az alábbiak szerint:

- mindegyik fejre *illesztjük* a korlátot (ez egyirányú egyesítés, hívásbeli változó nem kaphat értéket)
- 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 (egyszerűsítés vagy egypagáció esetén) kivesszük a tárból a kijelölt korlátokat, majd minden esetben végrehajtjuk a törzset.
- Ha ezzel 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 (az alvó passzív korlátok közé).

A futás akkor fejeződik be, amikor már nem marad aktiválható korlát. Az őr-részben (elvben) nem lehet változót érinteni. Az őr-rész két komponensből áll: `Ask & Tell`

- `Ask` — változó-érintés vagy behelyettesítési hiba meghiúsulást okoz
- `Tell` — nincs ellenőrzés, a rendszer „elhiszi”, hogy ilyen dolog nem fordul elő

9.3. A CHR szabályok szintaxisa

A SICStus kézikönyv alapján a CHR szabályok szintaxisa az alábbi szabályrendszerrel írható le:

```
Szabály       --> [Név @]
                    (Egyszerűsítés | Propagáció | Egypagáció)
                       [pragma Pragma].

Egyszerűsítés --> Fejek         <=> [Őr '|'] Törzs
Propagáció    --> Fejek         ==> [Őr '|'] Törzs
Egypagáció    --> Fejek \ Fejek <=> [Őr '|'] Törzs
```

```
Fejek          --> Fej | Fej, Fejek
Fej            --> Korlát | Korlát # Azonosító
Korlát         -->  egy korlátként deklarált meghívható kifejezés
Azonosító      -->  egy egyedi változó

Őr             --> Ask | Ask & Tell
Ask            --> Célsorozat
Tell           --> Célsorozat
Célsorozat     -->  egy meghívható kifejezés, konjunkciókkal és diszjunkciókkal

Törzs          --> Célsorozat

Pragma         -->  kifejezések konjunkciója, amik a #/2-vel azonosított fejekre hivatkoznak
```

Két fontosabb pragmát érdemes ismerni:

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

9.4. CHR példák

Az alábbi példa az $X \leq Y$ relációt valósítja meg CHR-ben.

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

handler leq.
constraints leq/2.

:- 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 $\leq$ reláció leírásához négy szabályt vezetünk be, mindegyik egy-egy elég triviális tulajdonságot ír le:

- Reflexivitás (`reflexivity`) — kifejezi, hogy ha a korlát-tárban van egy `X leq Y` korlát, és ugyanekkor az `X = Y` reláció teljesül, akkor a korlát triviálisan igaz, tehát helyettesíthetjük a `true` korláttal, azaz tulajdonképpen eltávolíthatjuk a korlát-tárból.

- Antiszimmetria (`antisymmetry`) — kifejezi, hogy ha a korlát-tárban egyszerre van jelen az `X leq Y` és az `Y leq X` korlát, akkor az csak úgy lehetséges, ha `X=Y`.
- Idempotencia (`idempotency`) — kifejezi, hogy ha a korlát-tárban kétszer szerepel az `X leq Y` korlát, akkor az egyiket eltávolíthatjuk.
- Tranzitivitás (`transitivity`) — az $X \leq Y \wedge Y \leq Z \Longrightarrow X \leq Z$ azonosság kifejezése.

Ezen négy szabály segítségével az `X leq Y, Y leq Z, Z leq X` célsorozatból a következőképpen jön rá a rendszer, hogy `X = Y` és `X = Z`:

Korlát-tár	Célsorozat	Tüzelés
—	`X leq Y, Y leq Z, Z leq X`	—
`X leq Y`	`Y leq Z, Z leq X`	—
`X leq Y, Y leq Z`	`Z leq X`	`transitivity` $\Longrightarrow$ `X leq Z`
`X leq Y, Y leq Z, X leq Z`	`Z leq X`	—
`X leq Y, Y leq Z, X leq Z` `Z leq X`		`antisymmetry` $\Longrightarrow$ `Z = X`
`X leq Y, Y leq X, X leq X` `X leq X, Z = X`		`reflexivity` kétszer
`X leq Y, Y leq X, Z = X`		`antisymmetry` $\Longrightarrow$ `Y = X`
`Y = X, Z = X`		

Nézzünk egy bonyolultabb CHR példát végeshalmaz-korlátokra! Az alábbi példa egy egyszerű `clpfd` keretrendszert valósít meg:

- két-argumentumú korlátokat kezel;
- a korlátokat egy (a keretrendszeren kívül megadott) `test/3` eljárás írja le:

  `test(C, X, Y)` sikeres, ha a `C` „nevű” korlát fennáll `X` és `Y` között;
- nem csak numerikus tartományokra jó.

```
handler dom_consistency.
constraints dom/2, con/3.
% dom(X,D) - az X változó a D listából veheti az értékeit
%           (D elemei ground-ok)
% con(C,X,Y) - a C korlát az X és Y változók között fennáll

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), % megpróbálja YD-t csökkenteni GY-nal
        (   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):- dom(X, DX),
          (   DX = [E] -> X = E
          ;   true
          ).

% labeling:
constraints labeling/0.

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

A `passive(Id)` úgynevezett `pragma` direktíva jelentése: az `#Id` jelöléssel hivatkozott fej csak passzív szerepű lehet. A fenti keretrendszer alkalmazásával az N királynő példa megoldása:

```
% Qs az N-királynő feladat megoldása
queens(N, Qs) :-
        length(Qs, N),
        make_list(1, N, L1_N),
        domains(Qs, L1_N),        % tartományok megadása
        safe(Qs),                 % korlátok felvétele
        labeling.                 % címkézés

% make_list(I, N, L): Az L lista az I, I+1, ..., N elemekből áll.
make_list(I, N, []) :- I > N, !.
make_list(I, N, [I|L]) :-
        I1 is I+1,
        make_list(I1, N, L).

% domains(Vs, Dom): A Vs-beli változók tartománya Dom.
domains([], _).
domains([V|Vs], Dom) :- dom(V, Dom), domains(Vs, Dom).

% queens(Qs): Qs egy biztonságos királynő-elrendezés.
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 által leírt királynőt, ahol I a Qs
% lista első elemének távolsága Q-tól.
no_attack([], _, _).
no_attack([X|Xs], Y, I) :-
```

```
        con(no_threat(I), X, Y),  % a korlát felvétele
        I1 is I+1,
        no_attack(Xs, Y, I1).

% "Az X és Y oszlopokban  I sortávolságra levő királynők nem
% támadják egymást" korlát definíciója, a dom_consistency
% keretrendszernek megfelelően
test(no_threat(I), X, Y) :-
        Y =\= X, Y =\= X-I, Y =\= X+I.

| ?- queens(4, Qs).
                                Qs = [3,1,4,2], labeling ? ;
                                Qs = [2,4,1,3], labeling ? ; no
```

Egy nem korlát-jellegű példa: az eratosztheneszi prímszita CHR-ben:

```
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 (ld. még `library('chr/examples/bool.pl')`):

```
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
```

Számosság-korlát megvalósítása 0-1 értékű változókra:

```
constraints card/4.

% L-ben a 1-ek száma >= A és =< B.
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).
% speciális esetek két változóra
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] , labeling ? ;
L = [1,0,1] , labeling ? ;
L = [1,1,_A] , labeling ? ;
L = [0,0,1,1] , labeling ? ;
L = [0,1,0,1] , labeling ? ;
L = [0,1,1,_A] , labeling ? ;
% ...
```

9.5. Egy nagyobb CHR példa kezdeménye

A feladat: adott egy négyzet, ahol bizonyos mezőkben egész számok vannak. A cél: minden mezőbe számot írni, úgy, hogy az azonos számot tartalmazó összefüggő területek mérete megegyezzék a terület mezőibe írt számmal.

A feladványt leíró adatstruktúra: `tf(Meret,Adottak)`, ahol `Meret` a négyzet oldalhossza, az `Adottak` egy lista, amelynek elemei `t(O,S,M)` alakú struktúrák. Egy ilyen struktúra azt jelenti, hogy a négyzet `S`. sorának `O`. oszlopában az `M` szám áll.

```
handler terulet.
constraints orszag/3, tabla/1, cimkez/0.

% orszag(Mezok, M, N): A Mezok mezőlista egy összefüggő, M méretű
% terület, amelynek kívánt mérete N. Egy mező Sor-Oszlop
% koordinátáival van megadva.

% tabla(Matrix): A teljes téglalap, listák listájaként.

% cimkez: Címkézési segédkorlát (ld. labeling az előző példákban)

foglalas(tf(Meret,Adottak), Mtx) :-
    bagof(Sor,
          S^bagof(Mezo,
                  O^tabla_mezo(Meret, Adottak, S, O, Mezo),
                  Sor),
          Mtx),
    append_lists(Mtx, Valtozok),          % listává lapítja Mtx-t
    MaxTerulet is Meret*Meret,
    domain(Valtozok, 1, MaxTerulet),
    tabla(Mtx),                           % tabla/1 korlát felvétele
    matrix_korlatok(Mtx, 1),              % orszag/3 korlátok
    cimkez.                               % címkézési segédkorlát

tabla_mezo(Meret, Adottak, S, O, M) :-
    between(1, Meret, S),                % 1..Meret felsorolása
    between(1, Meret, O),
    (   member(t(S,O,M), Adottak) -> true
    ;   true
    ).

matrix_korlatok([], _).
matrix_korlatok([Sor|Mtx], S) :-
    sor_korlatok(Sor, S, 1),
    S1 is S+1,
    matrix_korlatok(Mtx, S1).

sor_korlatok([], _, _).
sor_korlatok([M|Mk], S, O) :-
    orszag([S-O], 1, M),
    O1 is O+1,
    sor_korlatok(Mk, S, O1).
```

```
% Az orszag/3 korlátra vonatkozó szabályok

% Ha két ország szomszédos, és azonos számot tartalmaz, akkor
% össze kell vonni őket
orszag(Mezok1, H1, M), orszag(Mezok2, H2, M) <=>
                szomszedos_orszag(Mezok1, Mezok2) |
                H is H1+H2,
                M #>= H,
                append(Mezok1, Mezok2, Mezok),
                orszag(Mezok, H, M).

% Ha két ország szomszédos, és az egyik már pont annyi mezőből áll,
% ahányas számot tartalmaz, akkor a másik országban már nem szerepelhet
% ilyen szám
orszag(Mezok, M, M), orszag(Mezok1, _, M1) ==>
                szomszedos_orszag(Mezok, Mezok1) |
                M1 #\= M.

% Ha egy ország pont annyi mezőből áll, ahányas számot tartalmaz,
% akkor nem kell vele tovább foglalkozni (mivel a szabályok sorban
% tüzelnek, ezért ez mindig később tüzel, mint az előző)
orszag(Mezok, M, M) <=>
                true.

% Ha egy ország már nem terjeszkedhet, de kevesebb mezőből áll, mint
% ahányas szám van benne, akkor meghiúsulunk
orszag(Mezok, H, M), tabla(Mtx) ==>
                nonvar(M), H < M,
                \+ terjeszkedhet(Mezok, M, Mtx) | fail.

% Címkézési segédszabály
(orszag(Mezok, H, M) # Id1, tabla(Mtx) # Id2) \ cimkez  <=>
                fd_max(M, Max), H < Max |
                szomszedos_mezo(Mezok, Mtx, M), cimkez
                        pragma passive(Id1), passive(Id2).

% A Mezok mezőből álló, M méretű ország az Mtx táblában terjeszkedhet
terjeszkedhet(Mezok, M, Mtx) :-
    szomszedos_mezo(Mezok, Mtx, M0),
    fd_set(M0, Set), fdset_member(M, Set).

% Az Mk1 és az Mk2 mezőkből álló országok szomszédosak
szomszedos_orszag(Mk1, Mk2) :-
    member(S1-O1, Mk1), member(S2-O2, Mk2),
    (   S1 == S2 -> abs(O1-O2) =:= 1
    ;   O1 == O2, abs(S1-S2) =:= 1
    ).
```

```
% A Mezok országnak az Mtx mátrixban az M mező a szomszédja
szomszedos_mezo(Mezok, Mtx, M) :-
    member(S-O, Mezok),
    relativ_szomszed(S1, O1),
    S2 is S+S1, O2 is O+O1,
    non_member(S2-O2, Mezok),
    matrix_elem(S2, O2, Mtx, M).
    % A Mtx mátrix S2. sorának O2. eleme M.

relativ_szomszed(1, 0).
relativ_szomszed(0, -1).
relativ_szomszed(-1, 0).
relativ_szomszed(0, 1).

pelda(p1,  tf(5, [t(2,1,2),t(2,2,1),t(2,4,4),t(2,5,3),
                  t(3,4,2),t(4,2,5),t(4,4,3),t(5,1,3),
                  t(5,5,2)])).

pelda(p9,  tf(6, [t(1,1,1),t(2,3,1),t(2,6,4),t(3,1,3),t(3,6,3),
                  t(4,1,2),t(4,5,2),t(4,6,4),t(5,3,3),t(6,1,2),
                  t(6,5,3)])).

| ?- pelda(p1, _Fogl), foglalas(_Fogl, Mtx).
Mtx = [[2,4,4,3,3],
       [2,1,4,4,3],
       [3,5,5,2,2],
       [3,5,3,3,3],
      [3,5,5,2,2]],
cimkez,
tabla([[2,4,4,3,3],[2,1,4,4,3],[3,5,5,2,2],...]) ? ;
no
```

Irodalomjegyzék

[1] Jean-Charles Régin: A filtering algorithm for constraints of difference in csps. In Barbara Hayes-Roth – Richard E. Korf (szerk.): *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.* (konferenciaanyag). 1994, AAAI Press / The MIT Press, 362–367. p. ISBN 0-262-61102-3.
URL `http://www.aaai.org/Library/AAAI/1994/aaai94-055.php`.

[2] SWI-Prolog manual: library(clpfd): Constraint logic programming over finite domains. `http://www.swi-prolog.org/man/clpfd.html`, 2016.