Logično programiranje z omejitvami

Aritmetika v Prologu

V prologu računamo s števili takole:

```
?- X is (10 + 4) * 3.
X = 42.

?- X is 2 + 3, Y is 10 * X.
X = 5,
Y = 50.
```

Operator <code>is</code> sprejme spremenljivko `X` in aritmetični izraz `E`, izračuna vrednost izraza `E` in dobljeno vrednost priredi spremenljivki `X`.

Števila lahko tudi primerjamo z operatorji za primerjavo števil:

```
?- 221 * 19 =:= 247 * 17.
true.

?- 23 < 42.
true.
```

Takole bi implementirali funkcijo faktoriela:

```
faktoriela(0, 1).
faktoriela(N, F) :-
    N > 0,
    M is N - 1,
    faktoriela(M, G),
    F is N * G.
```

Preizkusimo:

```
?- faktoriela(7, F).
F = 5040.
```

V duhu prologa bi pričakovali, da `faktoriela` deluje v obse smeri:

```
?- faktoriela(N, 6).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [9] _13020>0
ERROR:    [8] faktoriela(_13046,6) at /var/folders/tw/2p25pzx951n_ht9607h1
ERROR:    [7] <user>
```

Napako se pojavi, ker `is` in < ne delujeta kot običajna predikata. V izrazu `X is E`, aritmetični izraz `E` ne sme vsebovati spremenljivk z neznano vredostjo. Na primer, če poskusimo izračunati `Y - Y`

```
?- Z is Y - Y.
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [8] _3132 is _3138-_3140
ERROR:    [7] <user>
```

dobimo napako, ker izraz na desni nima točno določene vrednosti. Tudi < ne deluje, če mu damo neznane vrednosti:

```
?- X < X + 1.
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [8] _5844<_5850+1
ERROR:    [7] <user>
```

Naj povemo, da predikat = ni uporaben pri računanju rezultatov, saj samo uporablja postopek združevanja:

```
?- X = 2 + 3.
X = 2+3.

?- 2 + X = 2 + 3.
X = 3.

?- X + 2 = 2 + 3.
false.

?- Z = Y - Y.
Z = Y-Y.
```

Danes bomo spoznali **logično programiranje z omejitvami** (angl. logic *constraint programming*), ki omogoča dosti bolj fleksibilno obravnavo aritmetičnih izrazov.

Logično programiranje z omejitvami

V logičnem programiranju je program spisek logičnih izjav, ki opisujejo rešitev (seveda morajo biti izjave izražene s Hornovimi formulami). V istem duhu bi lahko računanje s števili opisali z *enačbami* (in *neenačbami*) namesto z zaporedjem arimetičnih operacij in primerjav. Če bi prologu dodali algoritme za reševanje sistemov enačb (in neenačb), bi lahko takšne opise uporabili za računanje z aritmetičnimi izrazi.

V SWI prologu nam to omogoča knjižnica `clpfd` (constraint logic programming on finite domains):

```
?- use_module(library(clpfd)).
true.

?- Z #= Y - Y.
Z = 0,
Y in inf..sup.

?- 3 + X #= 3 + 2.
X = 2.

?- X + 2 #= 3 + 2.
X = 3.

?- 3 #< X, 4 * X #< 25.
X in 4..6.
```

Kot vidimo, je treba namesto operatorjev =, <, > uporabljati aritmetične omejitve #=, #<, #>, ... Ali lahko rešujemo tudi kvadratne enačbe?

```
?- X * X #= 1.
X in -1\/1.
```

Prolog je odgovoril, da je vrednost `X` bodisi `-1` bodisi `1`. Poskusimo še eno kvadratno enačbo:

```
?- X * X - 5 * X + 6 #= 0.
X in 2..sup,
5*X#=_30476,
X^2#=_30500,
_30476 in 10..sup,
-6+_30476#=_30500,
_30500 in 4..sup.
```

Tokrat smo dobili čuden odgovor. Poglejmo pobližje, kako deluje programiranje z omejitvami.

Domene

Programiranje z omejitvami vedno deluje na neki **domeni**, se pravi na množici vrednosti z dano strukturo. Tipične domene so:

- cela števila (domena `fd`)
- realna in racionalna števila (domena `qr`)
- Boolova algebra (domena `pb`)

Vsaka od teh domen zahteva specialne algoritme, ki znajo poenostavljati in združevati omejitve, ki so sestavni del programa. Mi bomo spoznali programiranje z omejitvami za cela števila, ki se imenuje tudi **končne domene** (angl. finite domain), ker vedno obravnavamo cela števila iz neke končne množice vrednosti.

Omejitve za domeno `fd`

V logičnem programiranju z omejitvami je program podan s Hornovimi formulami in **omejitvami**, ki predpisujejo dovoljene vrednosti spremenljivk. Prolog omejitve zbira in jih poenostavlja, z nekaj sreče pa jih kar razreši. Za domeno `fd` imamo več vrst omejitev.

Aritmetične omejitve

Aritmetične omejitve zapišemo z osnovnimi aritmetičnimi operacijami

```
+   -   *   ^   min   max   mod   rem   abs   //   div
```

in operatorji za primerjavo

```
#=   #\=   #>=   #=<   #>   #<
```

Intervalske omejitve

Intervalska omejitev

```
X in A..B
```

določi, da mora veljati `A ≤ X ≤ B`. Če želimo nastaviti samo zgornjo ali spodnjo mejo za `X`, lahko namesto `A` pišemo `inf` (kar pomeni -∞) in za `B` pišemo `sup` (kar pomeni +∞). Na primer,

```
X in inf..5
```

pomeni, da velja `X ≤ 5`. Pogosto želimo z intervalom omejiti več spremenljivk hkrati:

```
X in 1..5, Y in 1..5, Z in 1..5.
```

V tem primeru lahko uporabimo `ins`:

```
[X,Y,Z] ins 1..5.
```

V splošnem `L ins A..B` pomeni, da mora veljati `A ≤ X ≤ B` za vse elemente `X ∈ L` seznama `L`.

Kombinatorne in globalne omejitve

Omejitev `all_distinct([X₁, ..., Xᵢ])` zagotovi, da imajo spremenljivke X_1, …, X_i različne vrednosti. Primer uporabe bomo videli kasneje.

Ostale kombinatorne in globalne omejitve so opisane v priročniku za SWI prolog.

Naštevanje

Program z omejitvami napišemo v dveh delih:

1. Podamo omejitve.
2. Podamo zahtevo, da naj prolog našteje vse rešitve, glede na dane omejitve.

Drugi korak izvedemo s predikatom `label` (ter `indomain` in `labeling`, preberite sami). Če podamo samo omejitve, jih prolog izpiše, a ne pokaže konkretnih rešitev. Denimo, da želimo `X, Y ∈ {1, 2, 3, 4}` in `X < Y`:

```
?- [X,Y] ins 1..4, X #< Y.
X in 1..3,
X#=<Y+ -1,
Y in 2..4.
```

Prolog je omejitve poenostavil:

- `X ∈ {1, 2, 3}`
- `X ≤ Y - 1`
- `Y ∈ {2, 3, 4}`

Če dodamo še `label([X,Y])`, našteje konkretne rešitve:

```
?- [X,Y] ins 1..4, X #< Y, label([X,Y]).
X = 1,
Y = 2 ;
X = 1,
Y = 3 ;
X = 1,
Y = 4 ;
X = 2,
Y = 3 ;
X = 2,
Y = 4 ;
X = 3,
Y = 4.
```

Pogljemo si primere, s katerimi bomo nabolje spoznali, kako deluje programiranje z omejitvami.

Primer: faktoriela

Vrnimo se k funkciji faktoriela:

```
faktoriela(0, 1).
faktoriela(N, F) :-
    N > 0,
    M is N - 1,
    faktoriela(M, G),
    F is N * G.
```

Operatorje `is` in > zamenjajmo z omejitvami (in funkcijo preimenujmo, da ne bo zmede):

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

fakulteta(0, 1).
fakulteta(N, F) :-
    N #> 0,
    M #= N - 1,
    F #= N * G,
    fakulteta(M, G).
```

Opomba: obrnili smo vrstni red zadnjih dveh pogojev. S tem smo poskrbeli, da se *najprej* zabeležijo vse omejitve, šele nato pa se izvede rekurzivni klic.

Preizkusimo:

```
?- fakulteta(7, F).
F = 5040 .

?- fakulteta(N, 6).
N = 3.

?- fakulteta(N, 1).
N = 0 ;
N = 1 ;
false.
```

Primer: pitagorejske trojice

Pitagorejska trojica je trojica celih števil `(A, B, C)`, za katero velja `A² + B² = C²`. Poleg tega smemo zaradi simetrije med `A` in `B` predpostaviti `A ≤ B`.

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

pitagora(A, B, C) :-
    A #=< B,
    A * A + B * B #= C * C.

pitagora_do([A,B,C], N) :-
    pitagora(A,B,C),
    [A, B, C] ins 1..N,
    label([A,B,C]).
```

Primer: permutacije

Na vajah boste programirali permutacije seznamov v navadnem prologu. Permutacije števil lahko elegantno zapišemo z omejitvami:

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

```
permutacija(N, P) :-
    length(P, N),
    P ins 1..N,
    all_distinct(P).
```

V poizvedbi zahtevamo, da se rešitve dejansko naštejejo:

```
?- permutacija(6,P), label(P).
```

Pojasnimo vsako od zahtev:

- `length(P, N)` pove, da je `P` seznam dolžine `N`,
- `P ins 1..N` pove, da so vsi elementi seznama `P` števila med `1` in `N`
- `all_distinct(P)` pove, da so vsi elementi seznama `P` med seboj različni.
- `label(P)` je zahteva, da je treba našteti vse sesname `P`, ki zadoščajo navedenim trditvam.

Poskusimo:

```
?- permutacije(3,P).
P = [1, 2, 3] ;
P = [1, 3, 2] ;
P = [2, 1, 3] ;
P = [2, 3, 1] ;
P = [3, 1, 2] ;
P = [3, 2, 1].
```

Ker smo uporabili programiranje z omejitvami, zlahka računamo tudi "nazaj",

```
?- permutacije(N,[1,2,3]).
N = 3.
```

in preverimo, ali dana seznam je permutacija:

```
?- permutacije(N,[1,2,3,3]).
false.
```

Primer: Sudoku

V datoteki sudoku.pl je program, ki z omejitvami reši Sudoku. Skupaj ga poskusimo razumeti.

Načrt:

1. Imamo seznam vrstic `Rows`.
2. Imamo matriko 9×9:
   - imamo 9 vrstic: `length(Rows, 9)`.
   - vsaka vrstica ima dolžino 9: vsak element `Rows` je seznam dolžine `9`.
3. Vsi elementi matrike so med 1 in 9: vsak element `V` iz `Rows`, za vsak element `X` iz `V`, velja `X in 1..9`.

4. Vsaka vrstica je permutacija: vsak element `V` seznama `Rows` zadošča pogoju `all_distinct(V)`.
5. Vsak stolpec je permutacija.
6. Vsak podkvadrat 3 × 3 je permutacija.

Vsakego od zgornjih omejitev zapišemo v prologu.

Vsak element `Rows` je dolžine 9

Prvi poskus:

```
Rows = [X1,X2,X3,X4,X5,X6,X7,X8,X9],
length(X1,9),
length(X2,9),
length(X3,9),
length(X4,9),
length(X5,9),
length(X6,9),
length(X7,9),
length(X8,9),
length(X9,9).
```

Drugi poskus: uporabimo `maplist`.

```
length9(L) :- length(L,9).

maplist(length9, Rows).
```

Vsi elementi elemtov `Rows` so med 1 in 9

Prvi poskus:

```
Rows = [X1,X2,X3,X4,X5,X6,X7,X8,X9],
X1 ins 1..9,
X2 ins 1..9,
X3 ins 1..9,
X4 ins 1..9,
X5 ins 1..9,
X6 ins 1..9,
X7 ins 1..9,
X8 ins 1..9,
X9 ins 1..9.
```

Z uporabo `maplist`:

```
vsi_med_1_9(Row) :- Row ins 1..9.

maplist(vsi_med_1_9, Rows).
```

Z uporabo `append`:

```
append(Rows, Elementi), Elementi ins 1..9.
```

Vsaka vrstica je permutacija

```
map_list(all_distinct, Rows).
```

Vsak stolpec je permutacija

Ideja: matriko `Rows` transponiramo in zahtevamo, da so vrstice transponiranke permutacije:

```
transpose(Rows, Columns), map_list(all_distinct, Columns).
```

Vsak podkvadrat 3 × 3 je permutacija

Ideja: če imamo vrstice `P`, `Q`, `R`, lahko iz njih izluščimo kvadrat na levi:

```
P = [P1, P2, P3 | Ps]
Q = [Q1, Q2, Q3 | Qs]
R = [R1, R2, R3 | Rs]
K = [[P1, P2, P3],
     [Q1, Q2, Q3],
     [R1, R2, R3]]
```

Iz tega dobimo predikat, ki za vrstice `P`, `Q` in `R` pove, da so vsi trije kvadrati, ki tvorijo `P`, `Q`, `R`, permutacije:

```
kvadrati_permutacija([],[],[]).
kvadrati_permutacija(
   [P1, P2, P3 | Ps],
   [Q1, Q2, Q3 | Qs],
   [R1, R2, R3 | Rs]) :-
   all_distinct([P1, P2, P3, Q1, Q2, Q3, R1, R2, R3]),
   kvadrati_permutacija(Ps, Qs, Rs).
```