

Programowanie logiczne z ograniczeniami (więzami)

(ang. *Constraint Logic Programming*)

Artur Michalski
Informatyka II rok



Plan wykładu

- Problemy spełniania ograniczeń
- Motywacje na rzecz wprowadzenia programowania z ograniczeniami do logiki
- Programowania w logice z ograniczeniami na przykładzie skończonych dziedzin dyskretnych
- Kategorie dziedzin numerycznych w programowaniu logicznym z ograniczeniami



Problemy spełniania ograniczeń (ang. *Constraints Satisfaction Problems*)

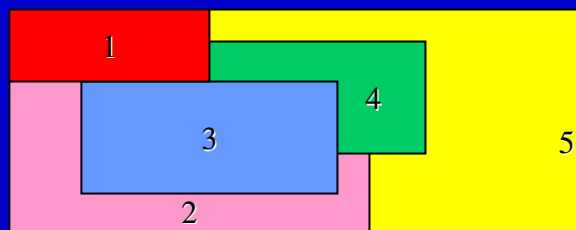
Problem spełniania ograniczeń opisujemy:

- Zbiorem zmiennych przedmiotowych
- Dziedzinami wartości wszystkich zmiennych
- Zbiorem ograniczeń, dotyczących zmiennych i określających zależności między nimi

Rozwiązaniem problemu spełniania ograniczeń jest taki zbiór podstawień zmiennych, który nie narusza nałożonych ograniczeń

Przykład problemu spełniania ograniczeń

Problem kolorowania mapy:



Cel: przypisać krajom kolory w takim sposób, aby żadne dwa sąsiednie kraje nie miały tego samego koloru

Zmienne: V_1, \dots, V_5 - kraje

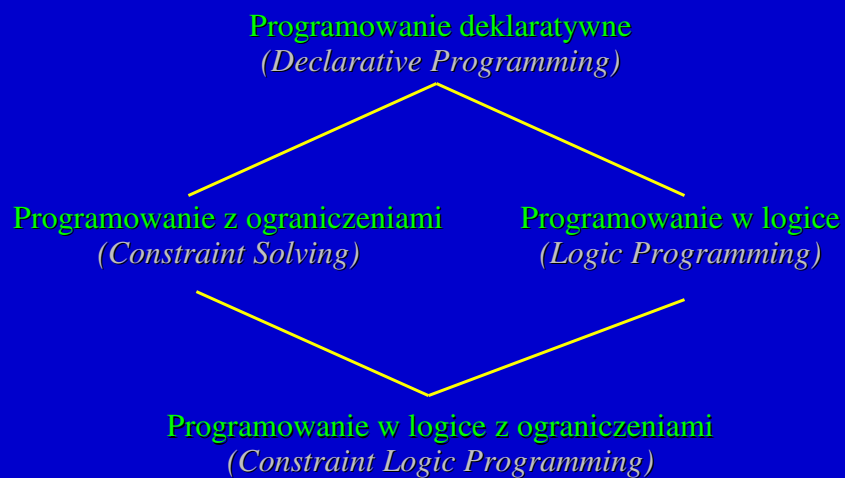
Dziedziny: D_1, \dots, D_5 : ze zbioru [red, blue, green, yellow, pink]

Ograniczenia: $adjacent(V_i, V_j) \Rightarrow V_i \neq V_j$

Gdzie występują problemy spełniania ograniczeń?

- Szeregowanie
- Harmonogramowanie
- Alokacja zasobów
- Trasowanie/Marszrutowanie
- Planowanie zadań

Deklaratywne programowanie w logice z ograniczeniami



Dlaczego programowanie w logice z ograniczeniami?

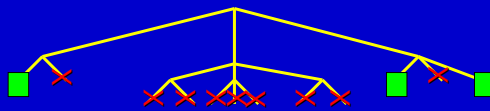
Motywacje

- Rozszerzenie paradygmatu programowania deklaratywnego na dziedziny liczbowe (numeryczne)
- Lepsza integracja przetwarzania numerycznego i metod logiki obliczeniowej w programowaniu deklaratywnym (inne podejście do operatorów arytmetycznych)
- Zwiększenie efektywności procesu wnioskowania (skuteczniejsze nawroty)

Efektywność programowania z więzami i programowania w logice

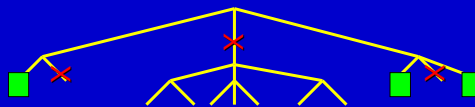
Programowanie w logice

„najpierw generuj, potem testuj”
duża liczba nieużytecznych ścieżek wnioskowania



Programowanie w logice z ograniczeniami (więzami)

„najpierw testuj, potem generuj”
wczesna eliminacja zbędnych ścieżek wnioskowania



Prolog - „generuj i testuj”

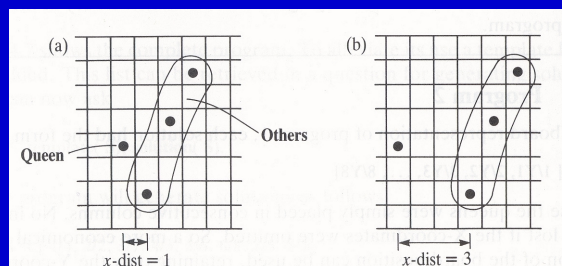
Przykład

Predykat `queens8 ([Y1, ..., Y8])` rozwiązuje problem 8 hetmanów i generuje prawidłowe położenia hetmanów `Y1, . . . , Y8`.

```
queens8 ([Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8]) :-  
    permut ([Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8], [1, 2, 3, 4, 5, 6, 7, 8]),  
    safe ([Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8]).  
  
permut ([], []).  
permut ([E|Perm], L) :- select (E, L, RL), % usuń E z listy L  
    permut (Perm, RL).  
  
safe ([]).  
safe ([Q|Rest]) :- noattack (Q, Rest, 1), safe (Rest).  
noattack (_, [], _).  
noattack (Q, [H|Rest], N) :- Q =\= H-N, Q =\= H+N,  
    N1 is N+1, noattack (Q, Rest, N1).
```

Prolog - „generuj i testuj”

Komentarz do predykatu `noattack (Queen, Others, Xdist)`:



- położenie hetmanów określone jest tylko przez ich współrzędną Y, a współrzędna X nie jest jawnie podana; wiemy jednak, że współrzędne Y hetmanów zawsze będą różne, zaś ich kolejność na liście jest istotna (bo odpowiada współrzędnej X)
- zatem, aby hetman **Queen** nie atakował innych hetmanów **Others**, odległość **Xdist** między nim a pierwszym z nich musi być różna od 1, między kolejnym różna od 2, itd.

Prolog - „generuj i testuj”

Komentarz do programu (1):

- a) reprezentacja rozwiązania opiera się na spostrzeżeniu, że skoro co najwyżej jeden hetman może znajdować się w każdym wierszu oraz w każdej kolumnie, to nie trzeba rozważać współrzędnych wiersza i kolumny, lecz wystarczy analizować permutacje zbioru elementów $\{1, 2, \dots, 8\}$ - wartość i -tego elementu permutacji jest numerem wiersza, w którym znajduje się hetman z i -tej kolumny
- b) metoda rozwiązania polega na generowaniu kolejnych permutacji powyższego zbioru elementów za pomocą predykatu **permut** a następnie testowaniu, czy nie narusza ona warunków sformułowanych w zadaniu (za pomocą predykatu **safe**)

Prolog - „generuj i testuj”

Komentarz do programu (2):

- c) rozwiązanie powyższe jest bardzo kosztowne; pierwsza sprawdzana permutacja $\{1, 2, 3, 4, 5, 6, 7, 8\}$ jest niepoprawna, bo hetmani stoją na głównej przekątnej; następuje nawrót i sprawdzenie permutacji $\{1, 2, 3, 4, 5, 6, 8, 7\}$, która też jest błędna!
- d) łatwo zauważyć, że wszystkie permutacje zaczynające się od $\{1, 2, \dots\}$ są niepoprawne, jednak program powyższy nie jest w stanie tego zweryfikować od razu; pierwsze prawidłowe rozwiązanie zostanie znalezione dopiero po analizie 2843 permutacji!!!

Prolog - „generuj i testuj”

Program powyższy można usprawnić, testując nie kompletne permutacje, lecz sprawdzając je już w trakcie ich generowania.

Predykat `queens8 (W, Closed, Open)` przechowuje na liście `Closed` pozycje wszystkich rozmieszczonych już hetmanów (w odwróconej kolejności), `Open` jest listą cyfr nie wykorzystanych jeszcze przy tworzeniu permutacji, listą `W` zaś zostanie przekazany wynik.

`queens8 (W) :-`

```
    queens8 (W, [], [1, 2, 3, 4, 5, 6, 7, 8]) .
```

`queens8 ([], _, []) .`

`queens8 ([H|Rest], Cl, Op) :-`

```
    select (H, Op, NewOp), % usuń H z listy Op
    noattack (H, Cl, 1),
    queens8 (Rest, [H|Cl], NewOp) .
```

◆ 314 ◆

Prolog - „generuj i testuj”

Komentarz do programu:

- a) rozwiązanie powyższe jest dużo sprawniejsze od poprzedniego - pierwszy poprawny wynik uzyskiwany jest już po sprawdzeniu 336 permutacji (zysk o jeden rząd wielkości!)
- b) nadal jednak nie jest w pełni wykorzystywane wiedza dostępna w trakcie rozwiązywania zadania - przykładowo, gdy pierwszy hetman znajdzie się w pierwszym wierszu, wiemy już, że cała główna przekątna jest zajęta, ale informacja ta jest wielokrotnie „odtworzana i odrzucana” w trakcie poszukiwania rozwiązania

◆ 314 ◆

Progr. w logice z ograniczeniami - „testuj i generuj”

Problem 8 hetmanów ma dwa ograniczenia:

- wartości współrzędnych muszą pochodzić ze zbioru liczb od 1 do 8
- po ustawieniu hetmana poniższe nierówności redukują liczbę dalszych dopuszczalnych ustawień reszty hetmanów:

$$\text{Hetman} = \backslash = \text{Następny},$$

$$\text{Hetman} = \backslash = \text{Następny} - N,$$

$$\text{Hetman} = \backslash = \text{Następny} + N$$

W systemach spełniania ograniczeń tworzone są niejawnie dodatkowe struktury danych, w których przechowuje się informację o możliwych wartościach zmiennych użytych w predykatkach i zgodnych w ograniczeniach.

Progr. w logice z ograniczeniami - „testuj i generuj”

- Początkowo zbiory wartości wszystkich zmiennych w problemie 8 hetmanów są takie same:

$$Y1 \in \{1,2,3,4,5,6,7,8\}$$

$$Y2 \in \{1,2,3,4,5,6,7,8\}$$

...

$$Y8 \in \{1,2,3,4,5,6,7,8\}$$

- Po ustawieniu pierwszego hetmana - przyjmijmy, że w 1-szym wierszu - następuje sprawdzenie ograniczeń i taka zmiana zbiorów wartości, która gwarantuje ich dalsze spełnienie:

$$Y1 \in \{1\} \qquad Y5 \in \{2,3,4,6,7,8\}$$

$$Y2 \in \{3,4,5,6,7,8\} \qquad Y6 \in \{2,3,4,5,7,8\}$$

$$Y3 \in \{2,4,5,6,7,8\} \qquad Y7 \in \{2,3,4,5,6,8\}$$

$$Y4 \in \{2,3,5,6,7,8\} \qquad Y8 \in \{2,3,4,5,6,7\}$$

Progr. w logice z ograniczeniami - „testuj i generuj”

- Rozstawienie kolejnych hetmanów w wierszach 3,5 oraz 7 spowoduje ograniczenie zbiorów wartości do dziedzin:

$Y1 \in \{1\}$	$Y5 \in \{2,4\}$
$Y2 \in \{3\}$	$Y6 \in \{4\}$
$Y3 \in \{5\}$	$Y7 \in \{2,6\}$
$Y4 \in \{7\}$	$Y8 \in \{2,4,6\}$
- Po krótkiej analizie widać, że nie istnieje kompletne rozwiązanie, będące rozszerzeniem powyższego i konieczny jest nawrót (brak bowiem wiersza o wartości 8)
- Utrzymywanie informacji o ograniczeniach, a nie ciągłe ich sprawdzanie i „zapominanie”, jak ma to miejsce w Prologu, pozwala zwiększyć o rząd wielkości szybkość wyszukiwania rozwiązania

Progr. log. z ograniczeniami w dziedzinie wartości dyskretnej i skończonych

Predefiniowane predykaty :

- Definiowania dziedzin wartości zmiennych
- Zapisu ograniczeń arytmetycznych
- Formułowania ograniczeń kombinatorycznych
- Sterowania procesem poszukiwania rozwiązania

Definiowane dziedziny wartości

- Operator **in** jest spełniony, gdy *zmienna* lub elementy *listy zmiennych* przyjmują wartości z podanego zakresu; służy do definiowania ograniczenia przynależności do zbioru wartości całkowitych

```
?- X in 1..4, [Y,Z] in 3..7, X=Y.  
X = {3..4}  
Y = {3..4}  
Z = {3..7}  
?- X in 1..3, Y in 4..7, X=Y.  
No
```
- Operator **in** dla *listy zmiennych* często występuje w postaci oddzielnego predykatu **domain (Vars, Min, Max)** lub **vars_in (Vars, Min, Max)**, które są równoważne zapisowi **Vars in Min..Max**, gdzie **Vars** oznacza listę zmiennych

Definiowane dziedziny wartości

- Zakresy wartości dla operatora **in** mogą być definiowane jako przedział **X..Y**, gdzie końce przedziału **X** i **Y** mogą przybierać wartości:
 - X** - zmienna lub liczba całkowita
 - min (X)** - najmniejsza wartość z dziedziny zm. **X**
 - max (X)** - największa wartość dziedziny zm. **X**
 - card (X)** - liczność dziedziny wartości zmiennej **X**
- albo na inne sposoby:
 - dom (X)** - jako dziedzina wartości zmiennej **X**
 - Range1 /\ Range2** - jako iloczyn dwóch zakresów
 - Range1 \/ Range2** - jako suma dwóch zakresów
 - \ Range** - jako dopełnienie zakresu

Definiowane dziedziny wartości

Przykłady definicji dziedzin wartości

```
?- X in 0..9, Y in dom(X).
```

```
X = {0..9}
```

```
Y = {0..9}
```

```
?- X in 0..4, Y in 5..9, Z in min(X)..max(Y).
```

```
X = {0..4}
```

```
Y = {5..9}
```

```
Z = {0..9}
```

```
?- X in 0..8, Y in 6..9, Z in dom(X) \ dom(Y).
```

```
X = {0..8}
```

```
Y = {6..9}
```

```
Z = {6..8}
```

◆ 3/14 ◆

Ograniczenia arytmetyczne

- Związki między zmiennymi i/lub wyrażeniami mogą być zapisywane jako ograniczenia arytmetyczne postaci:

$\langle \text{Expr} \rangle \langle \text{Op} \rangle \langle \text{Expr} \rangle$,

gdzie $\langle \text{Op} \rangle$ jest jednym z operatorów:

$\#< \#> \#=< \#>= \#= \#\backslash=$,

a $\langle \text{Expr} \rangle$ są dowolnymi wyrażeniami arytmetycznymi

```
?- X in 1..5, Z in 3..13, X+Y \#= Z.
```

```
X = {1..5}
```

```
Z = {3..13}
```

```
Y = {-2..12}
```

- Wyrażenia w ograniczeniach arytmetycznych nie muszą być liniowe - algorytm spełniania ograniczeń będzie jednak opóźniał ich spełnianie do czasu, gdy staną się liniowe

◆ 3/14 ◆

Ograniczenia arytmetyczne

- Ograniczenia arytmetyczne mogą dotyczyć całych list zmiennych, dzięki czemu ich zapis jest prostszy (unikamy definiowania ciągu ograniczeń elementarnych), a spełnianie mniej wymagające obliczeniowo
- Predykat **scalar_product** (**Cs**, **Vars**, **Op**, **Value**) jest spełniony, gdy iloczyn skalarny zmiennych z listy **Vars** i listy stałych **Cs** (tej samej długości) pełni relację **Op Value**
- Predykat **sum** (**Vars**, **Op**, **Value**) jest spełniony, gdy suma wartości zmiennych z listy **Vars** spełnia relację **Op Value**

Przykład

```
?- X in 0..9, Y in 1..2, sum([X,Y],#=#,2).  
X = {0..1}  
Y = {1..2}
```

Ograniczenia kombinatoryczne

Predykat **all_different** (**Vars**) jest spełniony, gdy wszystkie zmienne z listy **Vars** mają parami różne wartości; spełnienie weryfikowane jest dopiero podczas wiązania (uzgadniania) zmiennych z listy **Vars**

```
?- [X,Y] in 0..1, all_different([X,Y]).  
X = {0..1}  
Y = {0..1}  
?- [X,Y] in 0..0, all_different([X,Y]).  
No
```

Ograniczenia kombinatoryczne

Predykat `all_distinct (Vars)` jest spełniony, gdy zmienne z listy `Vars` o ukonkretnionych(określonych) i nieujemnych dziedzinach przyjmują różne wartości; predykat logicznie równoważny `all_different (Vars)`:

```
?- [X,Y,Z] in 0..1, all_dinstinct([X,Y,Z]).  
No  
?- [X,Y,Z] in 0..1, all_different([X,Y,Z]).  
X in {0..1}  
Y in {0..1} } ← bo zmienne nie zostały jeszcze związane!  
Z in {0..1}  
?- [X,Y,Z] in 0..1, all_different([X,Y,Z]), X=1.  
No } ← bo zmienna X została związana!
```

Sterowanie poszukiwaniem rozwiązania

Predykat `indomain (Var)` przypisuje zmiennej `Var` wartość z jej aktualnej dziedziny; wykonuje nawroty od najmniejszej możliwej wartości do największej

Przykład

Predykat `delnum (L, RL)` usuwa z listy `L` cyfry `i` w rezultacie otrzymujemy listę `RL`:

```
delnum(L, RL) :- X in 0..9, indomain(X),  
                 delete(L, X, RL). % usuwa wszystkie X z L  
?- delnum([a,1,b,0,c,0], RL).  
RL = [a,1,b,c]; } ← dla 0  
RL = [a,b,0,c,0]; } ← dla 1  
RL = [a,1,b,0,c,0]; } ← 8 razy, bo zostało 8 cyfr!  
...
```

Przeszukiwanie z **indomain** - problem 8 hetmanów

```
:- use_module(library('clp/bounds')).%SWI-Prolog
queens8([Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8]):-
    queens8([Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8],8).
queens8([],_).
queens8([Q|Qs],N):- [Q|Qs] in 1..N,
                    all_different([Q|Qs]),
                    indomain(Q),
                    noattack(Q,Qs,1),
                    queens8(Qs,N). %recursion
noattack(_,[],_).
noattack(Q,[H|T],N):- Q #\= H-N, Q #\= H+N,
                    N1 is N+1,
                    noattack(Q,T,N1).
```

Sterowanie poszukiwaniem rozwiązania

Predykat **labeling(Opt,Vars)** jest spełniony, gdy możliwe jest znalezienie dla wszystkich zmiennych z listy **Vars** takich podstawień wartości, które zachowują nałożone ograniczenia; parametr **Opt** jest 4-elementową listą predefiniowanych termów, które decydują o:

- strategii wyboru kolejnej zmiennej (**leftmost**, **min**, **max**, **ff**),
- strategii wyboru wartości dla zmiennej (**step**, **enum**, **bisect**),
- kierunku przeszukiwania dziedziny zmiennej (**up**, **down**),
- sposobie generowania rozwiązań (**all**, **minimize(X)**, **maximize(X)**).

```
?- [X,Y] in 0..1, labeling([], [X,Y]).
X = 0    Y = 0;
X = 0    Y = 1;
X = 1    Y = 0;
X = 1    Y = 1
```

Przeszukiwanie z **labeling** - problem 8 hetmanów

```
:- use_module(library('clp/bounds')).%SWI-Prolog
queens8([Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8]):-
    Qs = [Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8],
    Qs in 1..8, safe(Qs),
    all_different(Qs),
    labeling([],Qs). % no recursion!
safe([]).
safe([Q|Qs]):- noattack(Q,Qs,1), safe(Qs).
noattack(_,[],_).
noattack(Q,[H|Rest],N):- Q #\= H-N, Q #\= H+N,
    N1 is N+1,
    noattack(Q,Rest,N1).
```

◆ 3/14 ◆

Przykłady zastosowań programowania w logice z ograniczeniami

- Arytmograf
- Problem kolorowania mapy

◆ 3/14 ◆

Przykłady zastosowań programowania w logice z ograniczeniami

Predykat `sum(Arytm)` rozwiązuje zadanie z arytmografem SEND+MORE=MONEY; zwraca listę wartości unikalnych zmiennych `Arytm` występujących w zadaniu.

`sum(Arytm) :-`

```
Arytm = [S,E,N,D,M,O,R,Y], %unique vars only!  
Arytm in 0..9, S #> 0, M #> 0,  
all_different(Arytm),  
1000*S + 100*E + 10*N + D  
+ 1000*M + 100*O + 10*R + E  
#= 10000*M + 1000*O + 100*N + 10*E + Y,  
labeling([],Arytm).
```

?- `sum(L)`.

`L = [9, 5, 6, 7, 1, 0, 8, 2]`



Przykłady zastosowań programowania w logice z ograniczeniami

Problem kolorowania mapy

Każdemu z krajów trzeba przypisać jeden z czterech kolorów tak, aby żadne dwa sąsiadujące ze sobą kraje nie miały tego samego koloru. Istnieje twierdzenie, które gwarantuje, że zawsze jest to możliwe.

Opis mapy: predykat `nghb(Country, Neighbours)` np.:

`nghb(1, [2, 3, 5, 6])`.

`nghb(2, [1, 3, 4, 5, 6])`.

`nghb(3, [1, 2, 4, 6])`.

`nghb(4, [2, 3])`.

`nghb(5, [1, 2, 6])`.

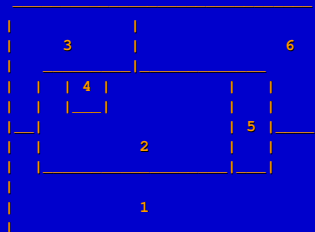
`nghb(6, [1, 2, 3, 5])`.



Przykłady zastosowań programowania w logice z ograniczeniami

Problem kolorowania mapy

Przykładowa mapa byłej Jugosławii



Opis mapy:

nghb (1, [2, 3, 5, 6]) .

nghb (2, [1, 3, 4, 5, 6]) .

nghb (3, [1, 2, 4, 6]) .

nghb (4, [2, 3]) .

nghb (5, [1, 2, 6]) .

nghb (6, [1, 2, 3, 5]) .

◆ 3/14 ◆

Przykłady zastosowań programowania w logice z ograniczeniami

Problem kolorowania mapy c.d.

Opis wyników:

Lista liczb reprezentujących kolory przypisane krajom:

[1, 3, 4, ...],

gdzie kolejność kolorów jest istotna i odpowiada kolejności w jakiej zdefiniowane zostały kraje za pomocą predykatu **nghb**.

Rozwiązanie:

Predykat **colors** (**Country_colour_list**) jest prawdziwy, o ile lista **Country_colour_list** spełnia ograniczenia narzucone przez relację **ngb** i treść zadania (m.in. cztery dostępne kolory reprezentowane przez zakres: 1 . . 4).

◆ 3/14 ◆

Przykłady zastosowań programowania w logice z ograniczeniami

Problem kolorowania mapy c.d.

Sformułowanie rozwiązania w Prologu:

```
colors (Colors) :- countries (C) ,
                    length (C, N) ,
                    length (Colors, N) ,
                    Colors in 1..4 ,
                    nghb_constr (N, Colors) ,
                    labeling ([], Colors) .

country (C) :- nghb (C, _) .
countries (CList) :- setof (C, country (C) , CList) .
```

◆ 3/14 ◆

Przykłady zastosowań programowania w logice z ograniczeniami

Problem kolorowania mapy c.d.

Pomocnicze predykaty:

```
nghb_constr (0, _) :- ! .
nghb_constr (N, Cs) :- nghb (N, L) ,
                        nth1 (N, Cs, Var) ,
                        adj_constr (Var, L, Cs) ,
                        N1 is N-1 ,
                        nghb_constr (N1, Cs) .

adj_constr (_, [], Cs) .
adj_constr (X, [A|As], Cs) :- nth1 (A, Cs, Var) ,
                              X #\= Var ,
                              adj_constr (X, As, Cs) .
```

◆ 3/14 ◆

Typy dziedzin w programowaniu logicznym z ograniczeniami

Constraint Logic Programming - CLP(x)

- CLP(clpr/R) - liniowe ograniczenia rzeczywistoliczbowe
- CLP(clpfd/FD) - dziedziny skończone (ang. FD-finite domain)
- CLP(clpb/B) - logika booleowska (algebra Boole'a)
- CLP(clpq/Q) - liczby wymierne (często razem z CLP(R) jako CLP(Q,R))
- CHR - Constraints Handling Rules

Programowanie w logice z ograniczeniami w Prologu

Zalety

- Rozszerzenie procesu unifikacji/uzgadniania na ogólniejsze mechanizmy spełniania ograniczeń
- Wprowadzanie ograniczeń różnych typów:
 - dziedzin skończonych: booleowskich, zbiorów skończonych, zbiorów uporządkowanych,...
 - dziedzin nieskończonych: liczb całkowitych, rzeczywistych,...
- Pojawienie się nowych metod rozwiązywania zadań z użyciem ograniczeń (efektywność!)