

Lezione23

Table of contents

- [Programmazione Logica con Vincoli](#)
 1. [Linguaggio di vincoli](#)
 1. [sat](#)
 2. [post](#)
 3. [label](#)
 2. [\\$CLP_0\(\mathcal{D}\)\\$](#)
 3. [\\$FD\\$](#)
 1. [Esempio interattivo](#)
 2. [Esempio non lineare interattivo](#)
 3. [SEND+MORE=MONEY](#)
 4. [n-Queens](#)
 4. [\\$CLP\(Dif\)\\$](#)

Programmazione Logica con Vincoli

L'argomento di cui parleremo in questo capitolo, unisce concetti già visti, soprattutto spiega meglio come mai abbiamo speso del tempo a parlare di CSP.

Uno dei problemi della programmazione logica è il fatto che calcoli non possono essere svolti, siccome tra l'insieme degli atomi non sono previsti nemmeno dei numeri. Per affrontare questo, introduciamo la **programmazione logica con vincoli** (CLP), dove "con vincoli" s'intende l'aggiunta di nuovi concetti che permettano di svolgere calcoli ma non solo.

Linguaggio di vincoli

La programmazione logica con vincoli è formata dalla programmazione logica e un **linguaggio di vincoli** a scelta, formato da

$$\langle D, C, A, V, sat, post, label \rangle$$

- un **dominio** del linguaggio dei vincoli D su cui andiamo a lavorare (come l'insieme dei numeri interi \mathbb{Z} o sottoinsieme);
- un insieme di **simboli di vincolo** C (come $=, >, <, \dots$) che permettano di fare affermazioni (come $x \geq 3$);
- un insieme di **variabili vincolate** V e un **insieme di atomi** A , che non è detto siano le stesse della programmazione logica;
- le asserzioni costruite usando atomi e vincoli, riguarderanno all'insieme dei domini tramite 3 funzioni a nome standard **sat** (satisfy), **post** (inviare) e **label** (etichettare).

Dato un linguaggio di vincoli, è possibile costruire tutti i CSP che vogliamo. Sapendo ciò, possiamo applicare tutte le nozioni passate imparate a questi nuovi problemi

sat

La funzione **sat** è una qualsiasi in grado di stabilire se il CSP è soddisfacibile o meno: in particolare, il risultato dell'applicazione della funzione a un CSP formato da elementi del nostro linguaggio dei vincoli è \perp se per certo insoddisfacibile, con tutti gli altri casi \top . Nell'ipotesi di scegliere un linguaggio di vincoli in cui **sat** termina sempre qualsiasi sia il CSP in esame, allora un risultato ci sarà a prescindere.

Che sia per davvero soddisfacibile o meno, un CSP, la funzione non è capace di determinarlo sempre: *sat* è una funzione che ci deve mettere velocemente nella condizione di andare avanti o meno.

post

La funzione post viene formalmente descritta come segue

$$post(\langle V_P, \{D\}, C_P \rangle, c(t_1, t_2, \dots, t_n)) = \langle V_P \cup vars(t_1, t_2, \dots, t_n), \{D\}, C_P \cup \{\gamma\} \rangle$$

dove V_P indica l'insieme di variabili, D il dominio e C_P i vincoli. Preso un CSP, *post* aggiunge un vincolo. All'aggiunta, viene generato un nuovo CSP con vincolo aggiuntivo con potenzialmente più variabili di prima e dominio invariato.

- Il vincolo viene costruito utilizzando un simbolo di vincolo c appartenente a C .
- Si costruiscono gli argomenti usando atomi e variabili, ottenendo una descrizione del vincolo.
- Il nuovo vincolo γ viene aggiunto al CSP.

label

La funzione non deterministica formalmente definita come

$$label(\langle V_P, \{D\}, C_P \rangle, x) = \begin{cases} (\langle V_P, \{D\}, C_P \rangle \cup \{\gamma\}, t) & \text{se il CSP ottenuto è soddisfacibile} \\ \perp & \text{altrimenti} \end{cases}$$

ha il compito di assegnare un valore possibile a una delle variabili del CSP, nel caso in cui il linguaggio dei vincoli ritenga che il problema sia risolubile. Viene fatto un passo dell'assegnamento parziale delle variabili, non sicuro che sia effettivamente valido ma che comunque può essere uno dei modi di risolvere il CSP. La funzione *label* è quella che per davvero ci porta a stabilire se c'è situazione soddisfacibile o meno, costruendo una soluzione, una variabile per volta, nell'ordine da noi indicato, tenendo attenzione al fatto che in alcuni linguaggi di vincoli, questa non termina.

≡ Esempio di linguaggio di vincoli

Considerando un linguaggio di vincoli con dominio $D = \mathbb{Z}$, considerando un insieme di atomi formato da un insieme di simboli, scriviamo il vincolo

$$x^2 - 4 \geq 0$$

con x simbolo di variabile, \geq simbolo di vincolo. Nel momento in cui abbiamo a disposizione una situazione simile, stiamo implicitamente facendo *post*, quindi stiamo verificando se il vincolo è soddisfacibile con *sat*, propagando il vincolo potenzialmente riducendo le possibilità e quindi soddisfacendo con

$$label(\langle \{x\}, \{\mathbb{Z}\}, \{c\} \rangle, x) = (P_L, t)$$

che in questo caso, in modo non deterministico, vale per tutti i termini associati al valore nell'insieme

$$\{z \in \mathbb{Z} : |z| \geq 2\} = \{\pm 2, \pm 3, \dots\}$$

Scritto in questo modo, possiamo notare che il CSP non terminerà mai.

CLP₀(\mathcal{D})

Se prendiamo un linguaggio di vincoli \mathcal{D} , formato dalle n -uple descritte all'inizio, possiamo costruire un'estensione di PROLOG₀ che chiamiamo CLP₀, che utilizza al suo interno questo linguaggio di vincoli per esprimere dei congiunti. Stiamo permettendo l'utilizzo dei simboli di vincolo come simboli di funzione di congiunti, permettendoci di esprimere $x \geq 0$ come goal per esempio: *sat* e *post* riescono a volte a definire se il goal è soddisfatto e se così non è, viene provata un'altra strada.

L'idea di questo linguaggio è quella di modificare la funzione σ_G : in questo caso, questa cerca nel programma un fatto o una testa che unifichino. Possiamo descrivere meglio la situazione pensando a come un goal di primo livello, venga soddisfatto in SWISH: cliccando "Run!" si parte da un CSP vuoto e man mano che incontriamo congiunti descritti con simboli di vincolo, *post* estenderà il CSP fintanto che *sat* o *post* non confermano che il goal non è soddisfacibile in tempo fissato; se il goal può essere soddisfatto e non abbiamo mai chiamato *label* (siccome da usare esplicitamente), vuole dire che le variabili del CSP sono legate tra loro con vincoli. PROLOG scriverà la variabile libera esplicitando i vincoli "sopravvissuti" all'operazione di propagazione.

Se la *label* viene chiamata su variabile, allora su questa verrà trovato un valore in tempo finito.

FD

Tra i linguaggi di vincoli più comuni c'è **finite domain** (*FD*).

Il `clpfd` è un'estensione di PROLOG che mette a disposizione il linguaggio dei vincoli *FD*, in cui le singole variabili sono variabili intere (niente \mathbb{R} , niente \mathbb{Q}) con segno. I simboli di vincolo che abbiamo a disposizione sono quelli normalmente a disposizione che per convenzione e non ambiguità, inizieranno sempre con un simbolo `#` (quindi scriveremo `x # ≥ 0` per esempio).

Il nome "finite domain" deriva dal fatto che la *label* applicata a tutte le variabili del CSP, garantisce se c'è soddisfacibilità o meno, unicamente nel caso in cui a forza di aggiungere vincoli, ogni variabile ha dominio sottoinsieme finito dei numeri interi \mathbb{Z} .

Se *label* viene utilizzata quando non siamo in condizione di certezza, un'eccezione viene lanciata.

Tradizionalmente, *FD* attua almeno un tipo di propagazione che è quello di bound consistency: quel tipo di propagazione che prende l'espressione aritmetica e la riduce sfruttando massimi e minimi degli intervalli. Viene prevista l'esistenza d'intervalli, che inizialmente non abbiamo ma che a seguito di accumulare vincoli, vengono costruiti.

Questa situazione è così comune che *FD* mette a disposizione dei vincoli che permettono di esplicitare il dominio: possiamo per esempio dire che la variabile `x` si trova tra `1` e `9` (sudoku).

Il senso dietro a questi operatori è quello di fornire un dominio finito su cui vogliamo che la variabile lavori, perché quello è l'obiettivo nostro. Mediante l'unione `\|` d'intervalli `..` si costruisce appunto l'unione d'intervalli, mentre mediante `in` (come `x in 1..10`) si va ad attribuire l'unione d'intervalli, al dominio della variabile a sinistra, che se non è singola ma piuttosto una lista di variabili, l'operatore va cambiato con `ins`.

Vincoli globali sono forniti. Almeno `all_distinct` c'è in questo gruppo, che prende una lista di variabili, imponendo che queste siano tutte diverse tra loro.

Il goal inserito all'inizio del programma

```
use_module(library(clpfd)).
```

carica l'intera libreria con tutte le particolarità che abbiamo visto.

≡ Calcolo della lunghezza di una lista

```
length([], 0).
length([_ | L], Length) :-
    Length # ≥ 0,
    N1 #= Length - 1,
    length(L, N1).
```

Una lista vuota ha lunghezza nulla (`length([], 0)`). La lista formata da una testa e un resto, avrà lunghezza data dalla lunghezza del resto + 1 (o riscritto `N1 #= Length - 1`). La lunghezza è sempre maggiore o uguale a 0 (`Length #= 0`). Con `#` stiamo indicando l'uso di vincoli di *FD*.

Arrivati a soddisfare un goal unificante con la testa, si comincia e procede con tutti i goal del corpo, il primo espresso con vincolo, per cui viene fatto *post*. Se la *sat* e la propagazione della *post* permettono di proseguire, siccome il CSP che stiamo costruendo è ancora soddisfacibile, seguiamo con il goal successivo, anche lui congiunto espresso come vincolo sul quale lo stesso procedimento viene applicato.

L'uso di *label* non viene fatto perché attribuirebbe un valore alla lunghezza delle liste, che implicitamente costruirebbe liste di lunghezza adeguata, ma lo farebbe a tentativi.

Esempio interattivo

La direttiva

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

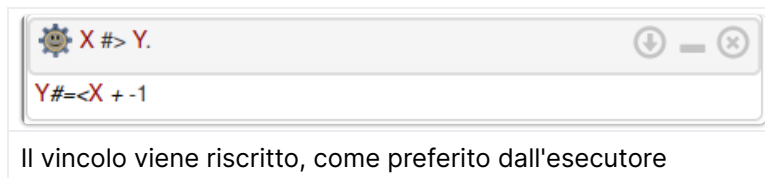
carica la libreria. Solitamente, il caricamento viene fatto all'interno del file di configurazione di un progetto, a nome `.swiplrc`, in modo d'assicurare che tutti i programmi lo utilizzino, siccome in un modo o nell'altro, così capita.

❗ Per informazioni e altri esempi: <https://github.com/triska/clpfd>

Nella casella dei goal in basso a destra, possiamo scrivere qualcosa come

```
X #> Y.
```

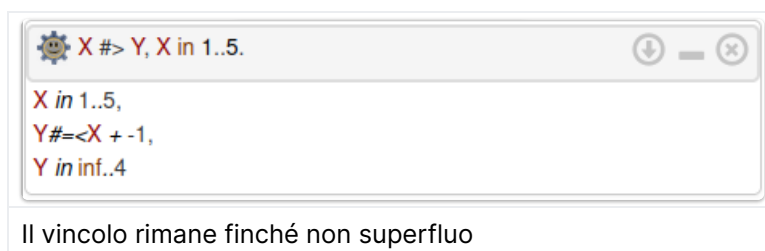
dove stiamo dicendo che qualsiasi sia *X* e qualsiasi sia *Y*, queste sono 2 variabili del linguaggio dei vincoli *CLP(FD)*, che possono quindi assumere valori interi.



Se scriviamo che *X* è compreso in `1..5` con

```
X #> Y, X in 1..5.
```

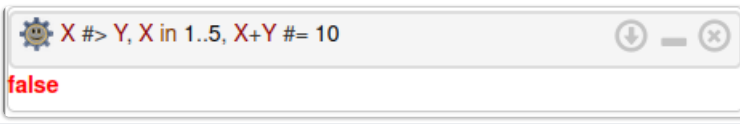
stiamo dando informazioni su *Y*: se *X* ha questi valori, *Y* non potrà essere maggiore di 4. Ci viene detto quale è il dominio di *X* (`in 1..5`) e il dominio di *Y* (`in inf..4`), dove bound consistency ci ha permesso di definire la seconda (`inf` indica $-\infty$, `sup` indica ∞).



Aggiungiamo un altro vincolo

```
X #> Y, X in 1..5, X+Y #= 10.
```

dove una copia X e Y non viene trovata, tale per cui la somma valga 10.

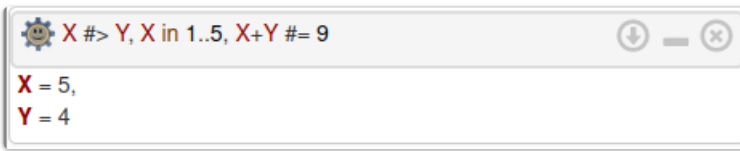


```
X #> Y, X in 1..5, X+Y #= 10
false
```

Se $X=5$ e $\max(Y)=4$, a 10 non ci arriviamo

Possiamo altrimenti vedere il caso in cui $Y \# = 9$, che restituisce invece un risultato positivo.

```
X #> Y, X in 1..5, X+Y # = 9.
```



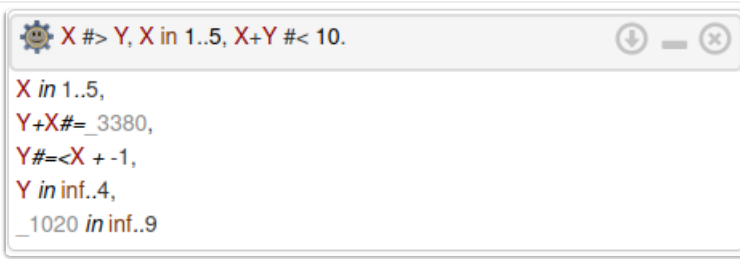
```
X #> Y, X in 1..5, X+Y # = 9
X = 5,
Y = 4
```

I valori massimi possibili per ciascuna variabile

Oppure possiamo vedere la situazione più interessante, siccome gli intervalli con cui stiamo lavorando, non sono intervalli finiti.

```
X #> Y, X in 1..5, X+Y #< 10.
```

Un'unica soluzione non esiste più, continuano a esserci vincoli. La situazione comincia ad essere complicata.



```
X #> Y, X in 1..5, X+Y #< 10.
X in 1..5,
Y+X#=_3380,
Y#=<X+-1,
Y in inf..4,
_1020 in inf..9
```


Il risultato al goal

Ci viene detto che X è compreso tra 1..5; $X + Y$ è una nuova variabile v in $\text{inf}..9$; la variabile Y è compresa tra $\text{inf}..4$. I vincoli non sono quelli che abbia scritto, venendo riformulati in un modo che per *FD* sia più semplice da manipolare.

In questi casi quando vogliamo attribuire un valore alla variabile e in modo non deterministico avere tutte le copie di assegnamenti che porteranno avanti soluzioni, usiamo *label* che se ha successo (*FD* attribuisce dominio finito a ogni variabile) allora in tempo finito una soluzione viene trovata.

A *label()* dobbiamo fornire sempre una lista di variabili a cui vogliamo attribuire i valori, nell'ordine che vogliamo. Per esempio, iniziamo con la variabile X , continuando poi con Y .

```
X #> Y, X in 1..5, label([ X, Y ]).
```


X #> Y, X in 1..5, label([X, Y]).

Arguments are not sufficiently instantiated
In:
[6] throw(error(instantiation_error,_1754))
[3] clpfd: '__aux_maplist/2_must_be_finite_fdvar+0' ([_1810]) at /usr/lib/swipl/library/clp/clpfd.pl:1794
[1] clpfd:labeling([],[_1862,_1868]) at /usr/lib/swipl/library/clp/clpfd.pl:1797


Note: some frames are missing due to last-call optimization.
Re-run your program in debug mode (:- debug.) to get more detail.

Il dominio non è finito per tutte le variabili

Con l'eccezione lanciata, ci viene detto che una variabile tra `X` e `Y` non è "abbastanza istanziata": i domini calcolati fino a un certo istante, non garantiscono che tutte le variabili abbiano un intervallo associato; la `label` non può funzionare. In verità, un intervallo finito lo abbiamo, possiamo provare a fare `label()` soltanto di `X` per verificarlo.

```
X #> Y, X in 1..5, label([ X ]).
```

Se `X=1` allora `Y` è compresa da `inf..0`; `label` è non deterministica e quindi cliccando "Next", ci verrà detto il risultato successivo.



X #> Y, X in 1..5, label([X]).

X = 1,
Y in inf..0
X = 2,
Y in inf..1
X = 3,
Y in inf..2
X = 4,
Y in inf..3
X = 5,
Y in inf..4

I risultati residuali

Proviamo a scrivere, per complicare la situazione, che `X` deve essere diverso da `3` con

```
X #> Y, X in 1..5, X #\=3, label([ X ]).
```


X #> Y, X in 1..5, X #\=3, label([X]).

X = 1,
Y in inf..0
X = 2,
Y in inf..1
X = 4,
Y in inf..3
X = 5,
Y in inf..4

X non può più essere uguale a 3

Esempio non lineare interattivo

Sempre usando la direttiva

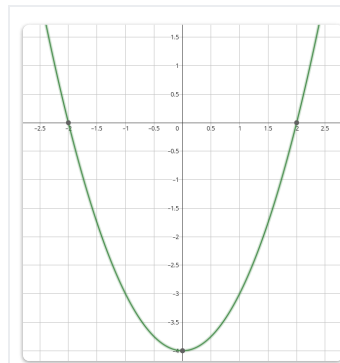
```
:- use_module(library(clpfd)).
```

vediamo un esempio non lineare. Se è tutto lineare, il sistema diventa uno di equazioni, disequazioni o inequazioni, risolvibile in modo sicuro. Se aggiungiamo cose non lineari, la situazione si complica.

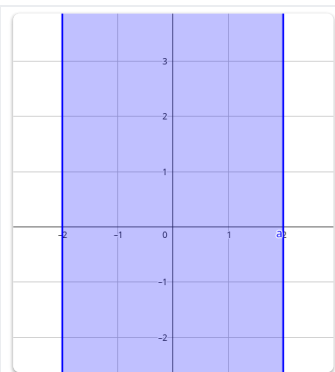
Con il seguente goal

$$X * X - 4 \# \leq 0.$$

stiamo descrivendo una parabola che interseca l'asse delle ascisse in -2 e 2 ; con \leq stiamo indicando l'intervallo contenuto tra le due soluzioni.



$$x^2 - 4$$



L'insieme delle soluzioni



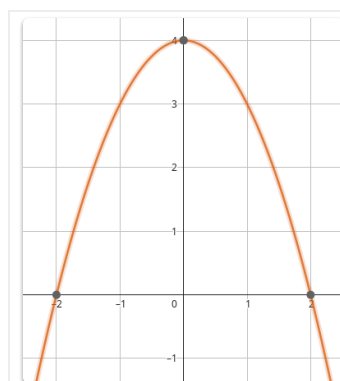
Il risultato al goal

Ci viene detto che in realtà X varia tra $-2..2$. Ci vengono dette diverse altre cose, che sono vincoli prodotti mano a mano che la funzione *post* ha propagato: la variabile 3194 varia tra $0..4$; la variabile interna che non vediamo 1022 varia tra $-4..0$.

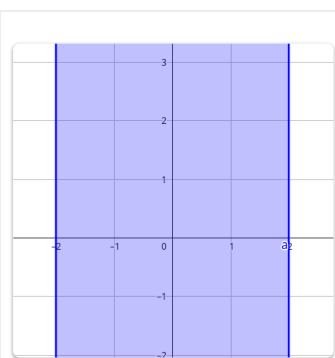
Sembra scontato, siccome la matematica dietro è semplice, ma vediamo ora un altro caso.

$$-X * X + 4 \# \geq 0.$$

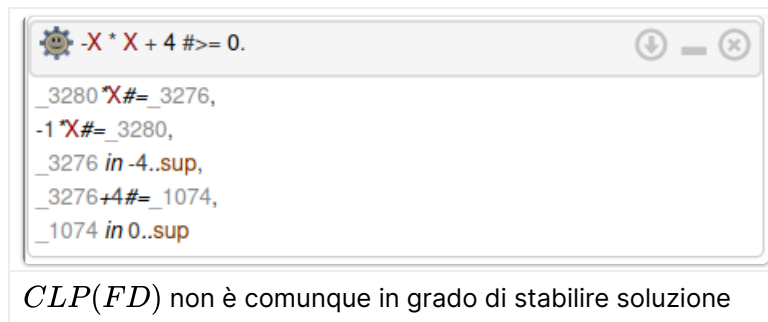
Qui abbiamo invertito i segni dell'equazioni, data la concavità verso il basso. Le soluzioni rimangono tuttavia le stesse, tra -2 e 2 .



$$-x^2 + 4$$



L'insieme delle soluzioni



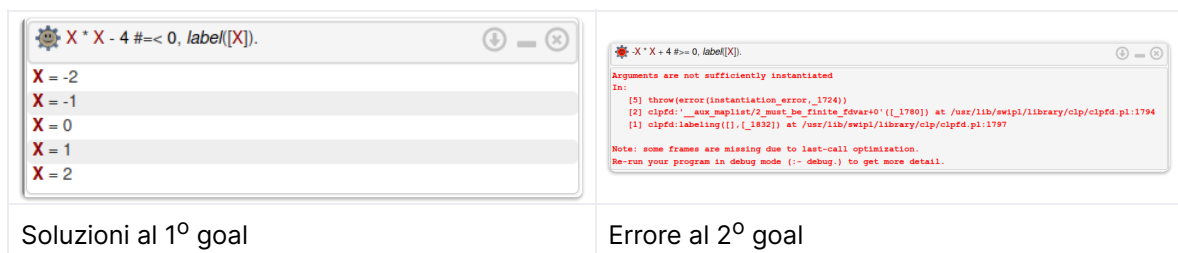
Nella soluzione vediamo che X non è compresa tra $-2..2$; se in questa situazione facciamo `label([X])`, nonostante le soluzioni siano identiche dal punto di vista matematico, FD non trova soluzione dato il modo in cui abbiamo scritto il goal.

Guardando le soluzioni con `label`

```
X * X - 4 #=< 0, label([X]).
```

```
-X * X + 4 #≥ 0, label([X]).
```

osserviamo che per il primo caso, il set di soluzioni viene trovato; per il secondo caso invece, un errore viene ritornato, come ci aspettavamo.



SEND+MORE=MONEY

Il seguente programma si riferisce allo stesso visto in [Lezione13](#).

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

send(Vars) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars in 0..9,
    M #\= 0, S #\= 0,
    all_distincts(Vars),
    S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    label(Vars).
```

Abbiamo un unico vincolo lineare con in aggiunta una sommatoria di variabili, a cui ciascuna è attribuito un peso. Ogni variabile varia tra $0..9$ e devono essere tutte diverse.

n-Queens

Il problema delle n -queens l'abbiamo già visto in [Lezione13](#).

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

queens(N, Queens) :-
    length(Queens, N),
```



```

Queens ins 1..N,
safe_queens(Queens),
label(Queens).

safe_queens([]).
safe_queens([Queen | QueenRest]) :-
    safe_queens(QueenRest, Queen, 1),
    safe_queens(QueenRest).

safe_queens([], _, _).
safe_queens([Queen | QueenRest], Queen0, D0) :-
    Queen0 #\= Queen,
    abs(Queen0 - Queen) #\= D0,
    D1 #= D0 + 1,
    safe_queens(QueenRest, Queen0, D1).

```

Utilizziamo un predicato ricorsivo, per andare a dire che la lista delle regine deve essere tale per cui 2 regine non hanno la stessa riga e colonna.

CLP(Dif)

CLP(FD) non serve unicamente per fare calcoli: serve per attribuire vincoli alle variabili per poi costruirci CSP su cui cercare la soluzione. Un altro linguaggio che abbiamo a disposizione nei PROLOG moderni è chiamato *CLP(Dif)*.

Dif è un linguaggio di vincoli che ha come dominio l'insieme dei termini, che permette di esprimere un unico vincolo: un unico simbolo di vincolo `dif` prende 2 termini, imponendo il vincolo che questi siano diversi.

```

:- use_module(library(dif)).

nevermember(_, []).
nevermember(X, [H | R]) :-
    dif(X, H),
    nevermember(X, R).

```

Se scriviamo `dif(X, H)`, stiamo imponendo il vincolo che la variabile `X` sia diverso dalla variabile `H`. Imporre che le due siano diverse vuole dire che il vincolo può essere imposto, in un istante, anche se le due variabili non hanno valore assegnato.