Creazione di nuovi vincoli

- In certi casi può essere utile creare nuovi vincoli
  - se il linguaggio non è abbastanza espressivo
  - se ho trovato un algoritmo efficiente per implementare un vincolo globale
- Metodi per implementare nuovi vincoli
  - implementazione sospensioni
  - vincolo **`element/3`**
  - libreria **`propia`**
  - Constraint Handling Rules

Sospensioni

- Vincoli gestiti internamente tramite il concetto di Sospensione. Una sospensione è un goal che può essere addormentato in attesa che avvenga un certo evento.
- Ricordate l'algoritmo AC3?
  - Due liste di vincoli: Lista dei vincoli attivi e lista di vincoli addormentati
  - Operazioni di: inserimento in lista, estrazione dalla lista, spostare vincoli da una lista all'altra (addormentare vincoli e risvegliare vincoli)
- ECLiPSe gestisce automaticamente le liste tramite uno scheduler.
- Noi dobbiamo solo addormentare un vincolo, in attesa di un evento. Quando l'evento si verifica, ECLiPSe mette il vincolo (goal) nel risolvente.
- Gli eventi nella libreria FD sono
  - min cancellazione del minimo nel dominio
  - max cancellazione del massimo
  - any cancellazione di un elemento qualsiasi del dominio
  - Ci sono altri eventi in altre librerie. Ad esempio, inst nella libreria suspend corrisponde all'istanziazione della variabile

AC3 (Mackworth)

La (List of active constraints) = lista di tutti i vincoli;
Ls (List of sleeping constraints) = ∅;
while La ≠ ∅ do
  prendi un vincolo $c(X,Y) \in$ La e toglilo da La
  se ci sono elementi inconsistenti in $dom(X)$
    allora eliminali (se $dom(X) = \emptyset$, fallisci)
    metti in La tutti i vincoli in Ls che coinvolgono X
  se ci sono elementi inconsistenti in $dom(Y)$
    allora eliminali (se $dom(Y) = \emptyset$, fallisci)
    metti in La tutti i vincoli in Ls che coinvolgono Y
  se $c(X,Y)$ non è completamente risolto
    allora mettilo in Ls

AC3 in ECLiPSe

La (List of active constraints) = lista di tutti i vincoli;
Ls (List of sleeping constraints) = ∅;
while La ≠ ∅ do
  prendi un vincolo $c(X,Y) \in$ La e toglilo da La
  metti in esecuzione il predicato $c(X,Y)$

ECLiPSe

Vincolo $c(X,Y)$ (utente)

  se ci sono elementi inconsistenti in $dom(X)$
    allora eliminali
  se ci sono elementi inconsistenti in $dom(Y)$
    allora eliminali
    metti in La tutti i vincoli in Ls che coinvolgono Y
  se $c(X,Y)$ non è completamente risolto
    allora sospendilo (mettilo in Ls)

  Se il dominio di X è stato modificato,
    metti in La tutti i vincoli in Ls che coinvolgono X
  Se il dominio di Y è stato modificato,
    metti in La tutti i vincoli in Ls che coinvolgono Y

Propagazione e sospensioni

- per implementare un nuovo vincolo (a basso livello) si
  - definisce un predicato
  - il predicato elabora i domini delle variabili coinvolte
  - se il vincolo non è completamente risolto, si sospende
- In pratica,
  - ECLiPSe implementa l'algoritmo AC3, con la lista dei vincoli sospesi (addormentati).
  - Noi possiamo implementare nuovi vincoli implementando la parte di propagazione (eliminazione di valori inconsistenti).

Primitive per gestire i domini

- dvar_domain(*X,D*) fornisce il dominio (dato astratto) della variabile *X*

  Es
  ```
  ?- X::1..10, dvar_domain(X,D).
          yes, D=1..10
  ```
- dom_to_list(*D,L*) trasforma il dominio in lista

  Es
  ```
  X::[1..3, 100..102], dvar_domain(X, D),
     dom_to_list(D,L).
          yes, D = [1..3, 100..102]
          List = [1, 2, 3, 100, 101, 102]
  ```

  Passa da una rappresentazione compatta ad una estesa: spesso sconsigliato

Primitive che forniscono informazioni sui domini

- dom_check_in(*+Element, +Dom*)
  - Verifica che l'intero *Element* sia nel dominio *Dom*.
- dom_compare(*?Res, +Dom1, +Dom2*)
  - Confronta due domini. *Res* viene unificato con
    - = sse *Dom1* = *Dom2*,
    - < sse *Dom1* ⊂ *Dom2*,
    - > sse *Dom2* ⊂ *Dom1*.
  - Fallisce se nessuno è sottoinsieme dell'altro
- dom_member(*?Element, +Dom*)
  - istanzia nondeterministicamente *Element* ad uno dei valori in *Dom*
- dom_range(*+Dom, ?Min, ?Max*)
  - Fornisce il minimo ed il massimo del dominio
- dom_size(*+Dom, ?Size*)
  - Fornisce il numero di elementi nel dominio

Primitive per eliminare valori

- dvar_remove_element(*+DVar, +El*)
  - Elimina *El* dal dominio della variabile *DVar*
- dvar_remove_smaller(*+DVar, +El*)
  - Elimina dal dominio di *DVar* gli elementi < *El*
- dvar_remove_greater(*+DVar, +El*)
  - Elimina dal dominio di *DVar* gli elementi > *El*
- dvar_update(*+DVar, +NewDom*)
  - Sostituisce il dominio della variabile *DVar* con *NewDom*. Il nuovo dominio deve essere un sottoinsieme del precedente, altrimenti → Errore

Operazioni sui domini

- **dom_copy(*+Dom1, -Dom2*)**
  - *Dom2* is a copy of the domain *Dom1*. Since the updates are done in-place, two domain variables must not share the same physical domain and so when defining a new variable with an existing domain, the domain has to be copied first.
- **dom_difference(*+Dom1, +Dom2, -DomDiff, ?Size*)**
  - The domain *DomDifference* is *Dom1* \ *Dom2* and *Size* is the number of its elements. Fails if *Dom1* is a subset of *Dom2*.
- **dom_intersection(*+Dom1, +Dom2, -DomInt, ?Size*)**
  - The domain *DomInt* is the intersection of domains *Dom1* and *Dom2* and *Size* is the number of its elements. Fails if the intersection is empty.
- **dom_union(*+Dom1, +Dom2, -DomUnion, ?Size*)**
  - The domain *DomUnion* is the union of domains *Dom1* and *Dom2* and *Size* is the number of its elements.
- **list_to_dom(*+List, -Dom*)**
  - Convert a list of ground terms and integer intervals into a domain *Dom*. It does not have to be sorted and integers and intervals may overlap.
- **integer_list_to_dom(*+List, -Dom*)** e **sorted_list_to_dom(*+List, -Dom*)**
  - simili a `list_to_dom`, V. differenze sul manuale

(meta)predicato Suspend

suspend(*+Goal, +Prio, +CondList*)

- Sospende il *Goal* in attesa che si verifichi una delle condizioni nella *CondList*
- *CondList* è una lista che contiene elementi del tipo

  Variabili -> *libreria*:*evento*
- *Prio* è una priorità da 1 a 12: vengono risvegliati prima i vincoli con priorità bassa
- Es:

```
suspend(c(A,B),3,[A->fd:min,B->suspend:inst])
```

sospende il goal `c(A,B)` e lo risveglia quando o viene eliminato il minimo nel dominio di `A` o viene istanziato `B`.

Sospensione di vincoli

- L'algoritmo AC3 utilizza una lista dei vincoli addormentati (o sospesi)
- In ECLiPSe ci sono più liste
  - Le liste sono associate alle variabili. In questo modo, quando si modifica il dominio di una variabile, si possono risvegliare tutti i vincoli associati a quella variabile
  - Per ogni variabile ci sono più liste; ogni lista è associata ad un evento
    - fd:min cancellazione del minimo nel dominio
    - fd:max cancellazione del massimo
    - fd:any cancellazione di un elemento qualsiasi del dominio
    - suspend:inst istanziazione della variabile
  - Le liste sono con priorità: ci sono vincoli con un algoritmo di propagazione veloce (che si vuole attivare spesso) e vincoli con algoritmo di propagazione più lento (che si vuole attivare meno spesso)

Esempio: implementazione di un vincolo

```
minimo(A,B,C):-
 dvar_domain(A,DomA),
 dom_range(DomA,MinA,MaxA),
 dvar_domain(B,DomB),
 dom_range(DomB,MinB,MaxB),
 min_int(MinA,MinB,MinMin),
 min_int(MaxA,MaxB,MaxMin),
 dvar_remove_smaller(C,MinMin),
 dvar_remove_greater(C,MaxMin),
 ( nonvar(A), nonvar(B), nonvar(C)
   ->true
   ; suspend(minimo(A,B,C),3,[A->fd:min,
       A->fd:max,B->fd:min,B->fd:max])
 ), wake.

min_int(A,B,B):- A>=B,!.
min_int(A,B,A):- A<B.
```

Estraggo i domini

Calcolo i nuovi domini

Elimino i val inconsistenti

Se vincolo risolto -> fine

Altrimenti sospendo

Risveglio altri vincoli

Esempio

- **A::3..5,B::2..6,C::0..9, minimo(A,B,C).**

```
A = A{[3..5]}
B = B{[2..6]}
C = C{[2..5]}

Delayed goals:
minimo(A{[3..5]}, B{[2..6]}, C{[2..5]})
```


Miglioramenti?

- ✗ Non faccio propagazione da **C** verso **A** e **B**

```
A :: 3..5, B :: 2..6, C :: 1..2, minimo(A, B, C).
A = A{[3..5]},B = B{[2..6]},C = 2
Delayed goals:
        minimo(A{[3..5]}, B{[2..6]}, 2)
```

- ✗ Risveglio il vincolo troppo spesso
  - Se **MinA<MinB** non c'è bisogno di svegliarsi su **B->fd:min**
- Se **MaxA<MinB**, so già che **A=C**:
  - potrei evitare di sospendere il vincolo **minore(A,B,C)** ed imporre il vincolo **A#=C**.

Domanda: che tipo di consistency ottengo? GAC? GBC?

Esercizio

- Si implementi il vincolo **abs_val(X,A)**, che impone che **A** sia il valore assoluto di **X**, tramite le sospensioni
- Si implementi la versione Bound-Consistency
- Si provi qual è la propagazione effettuata nei casi
  - **X:: -3..5, A :: -2..4, abs_val(X,A).**
  - **X:: -3..5,A ::[-4.. -2,1..4],abs_val(X,A).**
  - **X::[-3,0,3,6,9], A:: -9..9, abs_val(X,A).**
  - **X:: -9..9, A::[-3,0,3,6,9], abs_val(X,A).**

Esercizio 2

- Si implementi il vincolo **abs_val(X,A)**, che impone che **A** sia il valore assoluto di **X**, tramite le sospensioni
- Si implementi ora la versione Arc-Consistency e si provi la propagazione effettuata negli stessi casi:
  - **X:: -3..5, A :: -2..4, abs_val(X,A).**
  - **X:: -3..5,A ::[-4.. -2,1..4],abs_val(X,A).**
  - **X::[-3,0,3,6,9], A:: -9..9, abs_val(X,A).**
  - **X:: -9..9, A::[-3,0,3,6,9], abs_val(X,A).**

Nota

- I vincoli CLP(FD) non devono lasciare aperti dei punti di scelta

```
X::[-3,0,3,6,9], A::
-9..9, abs_val(X,A).

X = X{[-3, 0, 3, 6, 9]}

A = A{[0..9]}

Delayed goals:

abs_val(X{[-3,0,3,6,9]},
A{[0..9]})

Yes (0.00s cpu, solution
1, maybe more) ?  ;

No (0.00s cpu)
```

- I punti di scelta lasciati aperti si possono trovare con il tracer: un asterisco vicino a **EXIT** all'uscita da un predicato significa che quel predicato ha lasciato punti di scelta aperti

```
X::[-3, 0, 3, 6, 9], A :: -9..9, abs_val(X, A).
(5) 1 CALL abs_val(X{[-3, 0, ...]}, A{[-9..9]})%> creep
(6) 2 CALL dvar_domain(X{[-3,0,...]}, _676)   %> skip
(6) 2 EXIT dvar_domain(X{[-3,0,...]},[-3,0,3,...])%>skip
(17) 2 CALL dvar_remove_smaller(A{[-9..9]}, 0) %> skip
(17) 2 EXIT dvar_remove_smaller(A{[0..9]}, 0) %> skip
(19) 2 CALL dvar_domain(A{[0..9]}, _1652) %> skip
(19) 2 EXIT dvar_domain(A{[0..9]}, 0..9) %> skip
(21) 2 CALL propagate_a_to_x2(0,9,[-3,0,3,...],X{[-
3,0,...]})   %> skip
(21) 2 *EXIT propagate_a_to_x2(0,9,[-3,0,3,...],X{[-3,0,
...]})   %> skip
(26) 2 DELAY<3> abs_val(X{[-3,0,...]},A{[0..9]}) %>creep
(5) 1 *EXIT abs_val(X{[-3, 0, ...]}, A{[0..9]})   %>
```


Vincolo element/3

- Un vincolo è una relazione, quindi può essere scritto come tabella in cui elenco le coppie consistenti e quelle inconsistenti

X	Y	c(X,Y)
0	0	✓
0	1	✗
0	2	✓
1	0	✓
1	1	✗
1	2	✓

Vincolo element/3

- Oppure con una tabella in cui elenco solo le consistenti
- In questo caso, il vincolo è soddisfatto solo se viene selezionata una riga

X	Y
0	0
0	2
1	0
1	2

```
c(X,Y):-
  element(I,[0,0,1,1],X),
  element(I,[0,2,0,2],Y).
```


PROPIA

- Propia è una libreria per definire vincoli a partire da predicati (o trasformare predicati in vincoli)
- Considera le soluzioni possibili ed effettua la minima generalizzazione dei domini

```
c(0,0).
c(0,2).
c(1,0).
c(1,2).
C(2,4).
```

```
?- c(X,Y) infers most.
X = X{[0..2]}
Y = Y{[0, 2, 4]}
Delayed goals:
     c(X{[0..2]}, Y{[0, 2, 4]}) infers most
```

PROPIA

- Propia fornisce il metapredicato infers, che trasforma un predicato in vincolo. Dopo infers si possono usare varie parole chiave, che indicano il tipo di propagazione richiesto (AC è infers most o infers fd)
- Per effettuare la generalizzazione, utilizza il dominio delle variabili di un certo solver.
  - Si possono usare diversi solver (FD, Herbrand, …)
  - Bisogna aver caricato il solver corrispondente prima

```
lib(fd).
lib(propia).
```


Esempio equivalente

- Il predicato può essere un predicato generale, non è necessario che sia costituito da fatti
- L'importante è che in ciascuno dei casi definisca i domini delle variabili
- L'esempio di prima può essere riscritto:

```
c(X,Y):- X #>= 0, X #< 2, Y :: [0,2].
c(2,4).
```

```
?- c(X,Y) infers most.
X = X{[0..2]}
Y = Y{[0, 2, 4]}
Delayed goals:
     c(X{[0..2]}, Y{[0, 2, 4]}) infers most
```

PROPIA

- Si può usare anche con predicati non ground o ricorsivi

```
no_overlap(Start1,Dur1,Start2,Dur2):-
  Start1 #>= Start2+Dur2.
no_overlap(Start1,Dur1,Start2,Dur2):-
  Start2 #>= Start1+Dur1.

?- S::1..10, no_overlap(S,2,5,3) infers fd.
   S = S{[1..3, 8..10]}
   yes
```

Simile a disgiunzione costruttiva.

Compito 13 set 2007

- Un commesso viaggiatore deve passare per un insieme di città e poi tornare alla città iniziale, percorrendo meno chilometri possibile.
- Per ogni coppia di città collegate da una strada, è riportata la distanza in un insieme di fatti dista/3

```
dista(bologna,ferrara,50).
dista(ferrara,bologna,50).
dista(bologna,ravenna,84).
...
```

Modello semplice

- Lista di variabili:
  - La prima var è la prima città visitata
  - La seconda var è la seconda città visitata
  - …
- Domini: le varie città:
  - `[A,B,C,D…] :: [ferrara,ravenna,bologna…]`
- Vincoli: ci deve essere un arco fra una città e la successiva. Il vincolo è proprio il predicato **`dista`**: devo trasformare il predicato in vincolo
- Funzione obiettivo: min somma delle distanze

Consideriamo questo problema:

```
?- X :: 1..10000000,
   Y :: 1..10000000, X#>Y, Y#>X.
```


Soluzione?

- Noi ci accorgiamo immediatamente del fallimento, perché 'vediamo' i vincoli dall'esterno, non dall'interno
- Sappiamo che quando ci sono alcune combinazioni di vincoli, non ci possono essere soluzioni

Ordinamenti

- In matematica, i vincoli <, ≤, >, … sono associati agli ordinamenti
- Un ordinamento (parziale) è una relazione binaria che gode delle proprietà:
  - Riflessività: $a \leq a$.
  - Antisimmetria: $a \leq b, b \leq a \rightarrow a=b$
  - Transitività: $a \leq b, b \leq c \rightarrow a \leq c$.
- Quindi il vincolo in matematica è definito in base alle sue proprietà. Possiamo definire anche noi un vincolo basandoci sulle proprietà?

Constraint Handling Rules (CHR)

- Un modo per definire nuovi risolutori di vincoli.
- I vincoli sono definiti tramite delle regole, che possono essere di tre tipi: simplification, propagation e simpagation (caso particolare di simplification)

Propagation rules

In ECLiPSe, ci possono essere al massimo 2 vincoli nell'antecedente

- Una Propagation rule ha la sintassi:

```
c1, c2, … ==> guardia | body.
```

e significa che se i vincoli **c1, c2, …** sono nel constraint store e la **guardia** è vera, allora anche il **body** deve essere vero.

- La **guardia** è opzionale.
- Nella testa (antecedente) possono comparire solo vincoli. Questi sono i nuovi vincoli che vogliamo definire (non possono comparire vincoli predefiniti, come #<, alldifferent, …)
- Operazionalmente, se i vincoli **c1, c2, …** sono nello store, verifica se la **guardia** è vera, poi esegue il **body**.

Simplification rules

In ECLiPSe, ci possono essere al massimo 2 vincoli nell'antecedente

- Una Simplification rule ha la sintassi:

```
c1, c2, … <=> guardia | body.
```

e significa che se la **guardia** è vera, allora i vincoli **c1, c2, …** sono equivalenti al **body**.

- Nella testa (antecedente) possono comparire solo vincoli. Questi sono i nuovi vincoli che vogliamo definire (non possono comparire vincoli predefiniti, come #<, alldifferent, …)
- Operazionalmente, se i vincoli **c1, c2, …** sono nello store, verifica se la **guardia** è vera, poi toglie dal constraint store **c1, c2, …** ed esegue il **body**.

Simpagation rules

- Una Simpagation rule ha la sintassi:

```
c1 \ c2 <=> guardia | body.
```

ed è dichiarativamente equivalente a

```
c1 , c2 <=> guardia | body, c1.
```

però è più efficiente (il vincolo **c1** non viene prima tolto e poi ri-aggiunto)

- Operazionalmente, se i vincoli **c1** e **c2** sono nello store, verifica se la **guardia** è vera, poi toglie dal constraint store **c2** ed esegue il **body**.

Esempio: vincolo leq (less or equal)

```
reflexivity@  leq(X,X) <=> true.
antisymmetry@ leq(X,Y), leq(Y,X) <=> X=Y.
transitivity@ leq(X,Y), leq(Y,Z) ==>
  leq(X,Z).
```

```
leq(A,B), leq(B,C), leq(C,A)

leq(A,B), leq(B,C), leq(C,A), leq(A,C)

leq(A,B), leq(B,A), A=C

A=B, A=C
```


Note

- CHR non utilizza l'unificazione, ma il pattern-matching (fa l'unificazione solo in una direzione). Ad es, `leq(A,B)` non attiva la regola

  ```
  leq(X,X) <=> true.
  ```

  (la testa della regola deve essere più specifica)
- La guardia deve essere un semplice test (ad es, `var`, `ground`, …) non deve unificare variabili che compaiono nella testa. Se si effettuano unificazioni, la guardia fallisce. Ad es:

  ```
  leq(X,Y) <=> X=Y|true.
  ```

  è equivalente a `leq(X,X) <=> true.`
- Non posso usare nella testa delle regole dei vincoli predefiniti, ma solo vincoli definiti con CHR. Se voglio, posso scrivere delle regole che "trasformano" un vincolo CHR in un altro vincolo, tipo:

  ```
  leq(X,Y) ==> X#=<Y
  ```

  oppure

  ```
  leq(X,Y) <=> nonvar(X), nonvar(Y) | X =< Y.
  ```


CHR in ECLiPSe

- Per usare CHR, si deve caricare la libreria **`chr`**, oppure la nuova implementazione **`ech`** (V. sul manuale le differenze).

Libreria `chr` - uso

- La libreria **`chr`** prevede che ci sia un file .chr che contiene solo le regole CHR (non contiene codice Prolog)

```
[eclipse 1] lib(chr).
```

```
[eclipse 2] chr2pl(nomefile).
```

  - Legge il file nomefile.chr e lo trasforma in un file nomefile.pl

```
[eclipse 3] [nomefile].
```

  - Carica il file nomefile.pl

Libreria **ech** *- uso*

- La libreria **ech** è integrata nel Prolog, quindi si può mescolare codice Prolog e regole CHR nello stesso file .pl (o .ecl)
- Non è necessaria una compilazione separata
- La sintassi è leggermente diversa dalla libreria **chr**.

Sintassi - **lib(chr)**

solver.chr

```
handler nomeSolver.
constraints vincolo1/n1.
constraints vincolo2/n2, vincolo3/n3.
nomeRegola1 @ vincolo1(A,B,C), vincolo2(A,C) <=> guardia | vincolo3(C).
nomeRegola2 @ vincolo1(A,B,C), vincolo2(A,C) ==> guardia | vincolo3(A).
```

- Esempio:

leq.chr

```
handler less_or_equal.
constraints leq/2.
reflexivity@  leq(X,X) <=> true.
antisymmetry@ leq(X,Y), leq(Y,X) <=> X=Y.
transitivity@ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```


Sintassi - **lib(ech)**

solver.pl

```
:- lib(ech).
:- handler nomeSolver.
:- constraints vincolo1/n1.
:- constraints vincolo2/n2, vincolo3/n3.
nomeRegola1 ::= vincolo1(A,B,C), vincolo2(A,C) <=> guardia | vincolo3(C).
nomeRegola2 ::= vincolo1(A,B,C), vincolo2(A,C) ==> guardia | vincolo3(A).
```

- Esempio:

leq.pl

```
:- lib(ech).
:- handler less_or_equal.
:- constraints leq/2.
reflexivity ::=  leq(X,X) <=> true.
antisymmetry ::= leq(X,Y), leq(Y,X) <=> X=Y.
transitivity ::= leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```


Esercizio CHR

- Si crei, tramite CHR, un nuovo solver per variabili Boolean, con i vincoli **and**/3, **or**/3, **neg**/2, dove
  - **and(A,B,C)** significa che C = A /\ B
  - **or(A,B,C)** significa C = A \/ B
  - **neg(A,B)** significa B = not(A)
- Es: `[eclipse]` **and(A,B,C), neg(A,B).**

```
A = A, B = B, C = 0
Delayed goals:and(A, B, 0), neg(A, B)
yes
```