UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI
Corso di Laurea in Informatica

Tesi di Laurea Triennale

Definizione e trattamento del vincolo di cardinalità insiemistica nella libreria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Roberto Amadini

Anno Accademico 2006/2007

Ai miei genitori,
Bruno e Tiziana.

Ringraziamenti

Mi sembra doveroso, giunto a questo significativo momento della mia carriera scolastica, dedicare qualche riga alle persone che hanno contribuito più o meno direttamente al raggiungimento della laurea.
Il primo ringraziamento va quindi ai miei genitori, che in tutti questi anni hanno saputo sopportarmi e sostenermi, permettendomi di portare a termine il corso di studi senza mai farmi mancare nulla.
Un sentito ringraziamento va anche al relatore, il Prof. Gianfranco Rossi, che nonostante i mille impegni quotidiani è comunque riuscito a seguirmi nel lavoro di tirocinio e tesi con grande disponibilità e assoluta tranquillità.
E' stata inoltre fondamentale l'interazione con l'amico e compagno di corso Luca Lodi Rizzini, grazie alla quale è stato possibile superare anche gli ostacoli più ardui. Una citazione la meritano sicuramente anche Lucia e Cinzia per la disponibilità dimostrata nel passarmi tonnellate di appunti.
Non ultimo, mi sento in dovere di fare una piccola digressione per ricordare un amico scomparso troppo prematuramente... Ciao Enrico!

Indice

Capitolo 1

Introduzione

La **programmazione dichiarativa** (DP, *Declarative Programming*) è un paradigma di programmazione nel quale il programmatore descrive *cosa* il programma deve fare, piuttosto che *come* lo deve fare.
La DP è impiegata in vari ambiti dell'informatica tra i quali l'intelligenza artificiale, le basi di dati, la gestione di configurazione, i Web Services e la progettazione hardware. I principali vantaggi di questo paradigma riguardano la facilità di sviluppo e comprensibilità del programma, la sinteticità, la riusabilità e il parallelismo implicito.
I linguaggi dichiarativi possono essere sia *special-purpose* (come SQL, IDL, WSDL...) che *general-purpose*, i quali a loro volta possono dividersi in linguaggi funzionali (come Scheme e Haskell) e logici (come Mercury e Prolog).
Recentemente, si è cercato di far coesistere il paradigma DP in un contesto di programmazione tradizionale, in particolar modo **orientato agli oggetti** (OO, *object-oriented*).
Un possibile approccio al problema è quello di definire nuovi linguaggi di programmazione o estendere linguaggi già esistenti: è il caso di Alma-0, Singleton, DJ, Flow Java. Questa soluzione tuttavia comporta notevoli difficoltà di implementazione e di integrazione con sistemi pre-esistenti.
Un approccio alternativo al precedente è invece quello *library-based*: il paradigma DP è inserito in un linguaggio 'ospite' come una libreria dello stesso (quindi, tutte le istruzioni sono implementate utilizzando esclusivamente tale linguaggio).
Questa soluzione è sicuramente più adottata, soprattutto se il paradigma DP è in particolare quello della programmazione con vincoli.

La **programmazione con vincoli** (CP, *Constraint Programming*) può vedersi come una forma di DP che permette di manipolare esplicitamente vincoli (cioè *relazioni su opportuni domini* sia simbolici che numerici) che tro-

va numerose applicazioni pratiche nei piu' svariati ambiti (ad esempio bioinformatica, ottimizzazione, analisi finanziaria ecc...).
In particolare, se il linguaggio utilizzato per la programmazione con vincoli è di tipo logico si parla di **programmazione logica con vincoli** (CLP, *Constraint Logic Programming*).
Se inizialmente i linguaggi ospite utilizzati per 'inglobare' quelli con vincoli erano per lo più linguaggi logici, più recentemente sono state sviluppate librerie per il supporto della CP anche per i linguaggi OO più tradizionali come C++ e Java. Esempi di queste soluzioni sono ILOG, JSolver, Choco, Koalog, JACK, JCL.
Tuttavia, fino ad'ora poche proposte hanno esaminato il problema di estendere un linguaggio OO con una o più librerie che forniscano in modo esauriente il supporto alla DP general-purpose.

Il lavoro di tesi proposto è basato su **JSetL**, una libreria Java che offre funzionalità per il supporto della DP general-purpose simili a quelle generalmente utilizzate nella CLP: strutture dati logiche, unificazione, ricorsione, risoluzione di vincoli, non-determinismo.
In particolare, JSetL fornisce molte delle funzionalità offerte da CLP($\mathcal{SET}$) [1], un CLP nel quale il **dominio dei vincoli** è quello degli *insiemi*.

La nozione di **insieme** è una componente rilevante nella progettazione e sviluppo del software. Ciò nonostante, i linguaggi di programmazione 'tradizionali' forniscono un supporto molto limitato per il trattamento di questa struttura dati astratta.
Recentemente, svariati linguaggi di programmazione logica e funzionale general-purpose hanno introdotto un supporto alla gestione degli insiemi: è il caso ad esempio di CLPS, SuRE, CLAIRE e Conjunto.
Tali linguaggi, tuttavia, impongono *restrizioni* significative riguardo il trattamento del dominio degli insiemi.
Molto spesso infatti si richiede che gli insiemi siano *completamente specificati*, cioè non costituiti da elementi variabili. Inoltre, gli elementi di un insieme sono spesso confinati ad essere *atomici* ed *omogenei* in relazione ad un particolare dominio (ad esempio, interi o intervalli), impedendo di fatto la possibilità di gestire insiemi *annidati* ed *eterogenei*.

CLP($\mathcal{SET}$) è invece un linguaggio che permette di operare sugli insiemi con maggior flessibilità e generalità. Gli insiemi sono visti come *strutture dati primitive*, mentre le operazioni predefinite su di essi sono considerati *vincoli primitivi* del linguaggio.
Il dominio $\mathcal{SET}$ permette di trattare anche insiemi eterogenei, annidati e

parzialmente specificati in modo molto flessibile, preservando quindi la *dichiarativit*à della struttura generale CLP.
Si osservi che questa significativa caratteristica può comportare in alcuni casi un'elevata complessità computazionale: l'interesse è quindi principalmente rivolto alla *potenza espressiva* del linguaggio.

In questo lavoro di tesi definiremo all'interno di JSetL un particolare vincolo insiemistico: la **cardinalità** di insiemi **finiti**.
E' immediato osservare che se S è un insieme finito, la sua cardinalità (che indicheremo con $\#S$) è semplicemente il numero naturale corrispondente al numero di elementi di S.
Si noti però che il dominio $\mathcal{SET}$ non esclude che S sia *parzialmente specificato*: in questo caso vedremo che solamente una soluzione *dichiarativa* può portare (attraverso l'utilizzo del *non-determinismo*) alla completa risoluzione del vincolo di cardinalità (detto anche ***size***).
E' molto importante osservare che, al contrario di altri vincoli insiemistici *chiusi* al dominio degli insiemi (quali ad esempio *union*, *subset* ecc...), il vincolo *size* mette in relazione due strutture distinte: gli insiemi e gli **interi**.
Questa sarà la fonte principale dei problemi riguardanti la **soddisfacibilità** dei vincoli di cardinalità: attualmente il constraint solver di JSetL gestisce i vincoli sugli interi solamente in modo *parziale*.
Come si potrà notare, questa limitazione influenzerà notevolmente la risoluzione dei vincoli *size*.
Per cercare di superare questo problema, si farà ricorso ad apposite *tecniche di riscrittura* dei vincoli; ciò nonostante, si dimostrerà come questo accorgimento non sia sufficiente a garantire una risoluzione corretta di tutti i casi possibili.

Il lavoro di tesi è organizzato nel seguente modo:
Nel Capitolo 2 troveremo una piccola panoramica di JSetL che ne illustra le caratteristiche fondamentali.
Il Capitolo 3 sarà dedicato alla definizione del vincolo di cardinalità insiemistica e alle proprietà caratteristiche che esso deve soddisfare.
Nel Capitolo 4 si parlerà del trattamento parziale dei vincoli sugli interi da parte del solver di JSetL.
Il Capitolo 5 presenterà un sistema di operazioni sui vincoli atto a migliorare la risoluzione del vincolo *size* (e non solo...)
Nel Capitolo 6 verranno raccolti e formalizzati i risultati ottenuti per ottenere una nuova forma di risoluzione dei vincoli.
Il Capitolo 7 si occuperà invece (a grandi linee) della parte implementativa in codice Java (o pseudo tale).

Nel Capitolo 8 infine ci sarà spazio per conclusioni ed eventuali lavori futuri.

Capitolo 2

La libreria JSetL

JSetL è una libreria Java che combina la programmazione OO di Java con i concetti fondamentali della programmazione CLP (come variabili logiche, liste anche parzialmente specificate, unificazione, risoluzione di vincoli, non-determinismo) ed in particolare CLP($\mathcal{SET}$) (operazioni sul dominio degli insiemi come unione, intersezione, inclusione...).
La risoluzione dei vincoli in JSetL è inerentemente *non-deterministica*: l'efficienza non è l'obiettivo primario, si pone l'accento sull'efficacia (ciò può comportare anche una complessità computazionale elevata).
JSetL è stato sviluppato presso il Dipartimento di Matematica dell'Università di Parma da Elio Panegai, Elisabetta Poleo e Gianfranco Rossi; si tratta di un package completamente scritto in codice Java, importabile con la sola istruzione `import JSetL.*;`
La libreria è un software libero, re-distribuibile e/o modificabile sotto i termini della licenza GNU. La versione corrente è JSetL v1.3 (maggio 2007), tuttavia quella utilizzata nella tesi è leggermente modificata: il lavoro svolto è allineato all'integrazione della libreria proposta nella tesi di Delia Di Giorgio [6]. Ad ogni modo, tale variazione è limitata solamente all'interfaccia degli *insiemi logici* e non comporta stravolgimenti nell'utilizzo di JSetL .
Nei paragrafi successivi viene quindi presentata una breve panoramica della libreria che ne descrive le caratteristiche principali; per una visione più dettagliata si rimanda il lettore all'articolo [4].
Prima però vengono illustrate alcune notazioni simboliche utlizzate in seguito.

Notazioni 2.1. *Siano $C, C_1, \ldots, C_n$ generiche classi, $T, T_1, \ldots, T_n$ generici tipi ed $X, X_1, \ldots, X_n$ generiche variabili definite nel linguaggio Java. Allora,*

- *L'insieme di tutti i **tipi primitivi** di Java sarà indicato con* $\Pi \stackrel{def.}{=} \{\text{int, short, long, byte, char, boolean, float, double}\}$.

Il sottoinsieme di Π *degli* ***interi primitivi*** *di Java verrà invece indicato con* $I \stackrel{def.}{=} \{\text{int, short, long, byte}\}$.

- *Se* X *è una variabile di tipo primitivo (in particolare, intero), scriveremo che* $X \hat{\in} \Pi$ *(rispettivamente,* $X \hat{\in} I$*).*

- *Se analogamente* X *è un'****istanza*** *di una* ***classe*** C *(o di una sua classe* ***derivata****), scriveremo che* $X \hat{\in} C$

- *Definiamo un concetto generale di 'pseudo-unione':*
  $X \hat{\in} T_1 \hat{\cup} T_2 \ldots \hat{\cup} T_n$ *sse* $X \hat{\in} T_1 \vee X \hat{\in} T_2 \vee \ldots \vee X \hat{\in} T_n$

- *Se ad un oggetto* X *è assegnato il valore speciale* **null***, scriveremo che* $X = \perp$

- *Se* f *è una* ***funzione*** *che prende* n *variabili* $\langle X_1, \ldots, X_n \rangle$ *di tipo* T_i *con* $i = 1, \ldots, n$ *e restituisce un oggetto* $X \hat{\in} T$ *scriveremo che:*

$$f : T_1 \hat{\times} \ldots \hat{\times} T_n \longmapsto T \text{ ed } X = f(X_1, \ldots, X_n)$$

  Si noti che f potrebbe anche essere ***parziale****, cioè potrebbe accadere che* $f(X_1, \ldots, X_n) = \perp$.
  In particolare, se $f : T \longmapsto T$*, preso* $k \in N$ *posso applicare* $\overbrace{f(f(\ldots(f}^{k \text{ volte}}(X))\ldots)$ *a patto che* $\overbrace{f(f(\ldots(f}^{i \text{ volte}}(X))\ldots) \neq \perp$ *per* $i = 1, \ldots, k$.

$$\text{Definiamo quindi } f^{(k)}(X) \stackrel{def.}{=} \overbrace{f(f(\ldots(f}^{k \text{ volte}}(X))\ldots)$$

- *Indicheremo con* $\mathcal{LV}$ *la classe* Lvar *(appartenente al package JSetL)*

- *Indicheremo con* $\mathcal{LL}$ *la classe* Lst *(appartenente al package JSetL)*

- *Indicheremo con* $\mathcal{LS}$ *la classe* LSet *(appartenente al package JSetL)*

- *Indicheremo con* $\mathcal{I}$ *la struttura degli* Interi*, a cui appartengono istanze della classe* Integer *oppure interi di tipo primitivo:*
  $\mathcal{I} \stackrel{def.}{=} \text{Integer } \hat{\cup } I$

- *Indicheremo con* $\mathcal{O}$ *la struttura degli* Oggetti*, a cui appartengono istanze della classe* Object *oppure di un qualsiasi tipo primitivo:*
  $\mathcal{O} \stackrel{def.}{=} \text{Object } \hat{\cup } \Pi$

- *Indicheremo con* Σ *la classe* String

2.1 Strutture dati logiche

Definizione 2.2 (Struttura Dati Logica)**.** *Sia C una delle seguenti strutture dati (classi):*

- *Variabile logica*
- *Lista logica*
- *Insieme logico*

Allora C è una ***struttura dati (classe) logica*** *e un istanza X di tale classe è detta* ***oggetto logico****.*
Ad ogni oggetto logico X può essere eventualmente assegnato un ***valore relativo*** *v = val(X) e un* ***nome esterno*** *s = ext(X) dove v è un qualsiasi valore (di tipo primitivo o discendente dalla classe* Object*) e s è di tipo* String*.*
Formalmente, una classe logica è una classe $C \in \{\mathcal{LV}, \mathcal{LL}, \mathcal{LS}\}$ *e un oggetto logico è un oggetto* $X \hat{\in} C$*. La struttura* $\mathcal{LV} \hat{\cup} \mathcal{LL} \hat{\cup} \mathcal{LS}$ *d'ora in avanti sarà denotata con* $\mathcal{L}$*.*
Inoltre, val e ext sono due funzioni:

$$val : \mathcal{L} \longmapsto \mathcal{O} \qquad\qquad ext : \mathcal{L} \longmapsto \Sigma$$

Se il nome esterno non è definito $ext(X) =$ `Id` *dove* `Id` *è un identificativo univoco assegnato di default.*
Se il valore relativo non è definito, si ha che $val(X) = \perp$*.*

Il nome esterno è una stringa che può risultare utile in fase di stampa dell'oggetto. Il valore relativo dell'oggetto può essere di tipo primitivo oppure un oggetto di una qualsiasi altra classe: in particolare, **può essere a sua volta un oggetto logico**.
Siano quindi $X \hat{\in} \mathcal{L}$ *e* $X_1 = val(X)$. Se $X_1 \hat{\in} \mathcal{L}$, posso considerare
$X_2 = val(X_1) = val^{(2)}(X)$. A sua volta, se $X_2 \hat{\in} \mathcal{L}$, posso considerare
$X_3 = val(X_2) = val^{(2)}(X_1) = val^{(3)}(X)$ e così via...
Dopo un certo numero finito n di iterazioni, troverò quindi un valore
$X_n = val^{(n)}(X)$ *t.c.* $X_n \hat{\notin} \mathcal{L}$.

Definizione 2.3 (Oggetto Logico Non-Inizializzato)**.** *Un oggetto logico X si dice* ***non-inizializzato*** *(o* ***unknown****) se il suo valore non è definito oppure è un oggetto logico che fa riferimento ad un valore non definito. Formalmente, un oggetto* $X \hat{\in} \mathcal{L}$ *è unknown se:*

$$(\exists\, n \in N \setminus \{0\})\ val^{(n)}(X) = \perp$$

In caso contrario, l'oggetto logico è detto ***inizializzato*** *(o* ***known****).*
Grazie a tale definizione, posso allora considerare la ***funzione booleana*** $\kappa : \mathcal{L} \longmapsto \{0,1\}$ *tale che:*

$$\kappa(X) = \begin{cases} 1 & \text{se } X \text{ è known} \\ 0 & \text{se } X \text{ è unknown} \end{cases}$$

La funzione $\kappa(X)$ *è implementata in JSetL dal metodo pubblico* `X.isknown()`

Definizione 2.4 (Variabile Logica)**.** *Una* ***variabile logica*** *è un oggetto logico che è istanza della classe* `Lvar`*, creato con la dichiarazione*

```
Lvar nomeVar = new Lvar(nomeExt,valRel);
```

dove `nomeVar` *è il nome della variabile,* `nomeExt` *il nome esterno (opzionale) e* `valRel` *il suo valore relativo (opzionale).*
Formalmente abbiamo che

$\texttt{nomeVar} \hat{\in} \mathcal{LV} \qquad \texttt{nomeExt} = ext(\texttt{nomeVar}) \qquad \texttt{valRel} = val(\texttt{nomeVar})$

Il valore relativo di una variabile logica può essere specificato al momento della costruzione ma in seguito **non può essere modificato direttamente** con metodi della classe `Lvar`: la sua manipolazione può avvenire solamente attraverso il meccanismo di risoluzione dei vincoli (vedi sezione 2.2). Inoltre, la classe `Lvar` fornisce metodi di utilità con i quali è possibile leggere o stampare il valore relativo, il nome esterno, conoscere se una variabile logica è inizializzata e così via...

Definizione 2.5 (Lista Logica)**.** *Una* ***lista logica*** *è un oggetto logico che è istanza della classe* `Lst`*, creato con la dichiarazione*

```
Lst nomeLst = new Lvar(nomeExt,valRel);
```

dove `nomeLst` *è il nome della lista,* `nomeExt` *il nome esterno (opzionale) e* `valRel` *è il suo valore relativo (opzionale), ossia la* ***collezione dei suoi elementi*** $c_1, \ldots, c_n$*.*
Formalmente abbiamo che

$\texttt{nomeLst} \hat{\in} \mathcal{LL} \qquad \texttt{nomeExt} = ext(\textbf{nomeLst}) \qquad \texttt{valRel} = val(\texttt{nomeLst})$

*La **lista vuota** è denotata dalla costante* `Lst.empty` *e verrà indicata con* [ ].
Se una lista contiene n elementi $c_1, \ldots, c_n$ *verrà indicata con* $[c_1, \ldots, c_n]$.
Inoltre, $[c_1, \ldots, c_n|R]$ *con R lista, denota la **concatenazione** delle liste* $[c_1, \ldots, c_n]$ *ed R. Se in particolare R è unknown,* $[c_1, \ldots, c_n|R]$ *è una **lista illimitata (unbounded)** in quanto i primi n elementi* $c_1, \ldots, c_n$ *sono noti, ma gli elementi di R non lo sono (si noti che potrebbe essere anche che R =* [ ]*).*

Definizione 2.6 (Insieme Logico)**.** [1] *Un **insieme logico** è un oggetto logico che è istanza della classe* `ConcreteLSet`*, creato con la dichiarazione*

```
LSet nomeSet = new ConcreteLSet(nomeExt,valRel);
```

dove `nomeSet` *è il nome dell'insieme,* `nomeExt` *il nome esterno (opzionale) e* `valRel` *è il suo valore relativo, ossia la collezione dei suoi elementi (opzionale). Formalmente abbiamo che*

$$\texttt{nomeSet} \hat{\in} \mathcal{LS} \qquad \texttt{nomeExt} = ext(\texttt{nomeSet}) \qquad \texttt{valRel} = val(\texttt{nomeSet})$$

*L'**insieme vuoto** è denotato dalla costante* `LSet.empty` *e verrà indicato con* { }.
Se un insieme contiene n elementi $c_1, \ldots, c_n$ *verrà indicato con* $\{c_1, \ldots, c_n\}$.
Inoltre, $\{c_1, \ldots, c_n|R\}$ *con R insieme, denota l'**unione insiemistica*** $\{c_1, \ldots, c_n\} \cup R$. *Se in particolare R è unknown,* $\{c_1, \ldots, c_n|R\}$ *è un **insieme illimitato (unbounded)** in quanto gli elementi* $c_1, \ldots, c_n$ *sono noti ma gli elementi di R non lo sono (si noti che potrebbe essere anche che R =* { } *).*

La differenza fondamentale tra liste e insiemi consiste nell'*ordine* (per le liste, al contrario degli insiemi, si può parlare di *i-esimo elemento*) e nella presenza di *elementi ripetuti* (gli insiemi non ammettono tali ripetizioni, al contrario

[1] Questa non è l'unica definizione possibile di insieme logico, potrei avere una definizione più generica del tipo
`MutableLSet nomeSet = new ConcreteMutableLSet(nomeExt,valRel);` che mi permetta di sfruttare sia le funzionalità degli insiemi di Java (`java.util.Set`) che quelle degli insiemi logici 'propri' di JSetL (`LSet`). Tuttavia, in questo contesto è sufficiente limitarsi alla definizione proposta.

delle liste).
Gli insiemi e le liste logiche sono collezioni di elementi di **qualsiasi tipo**. Questo significa che tali strutture possono contenere anche altri oggetti logici; in particolare, possono a loro volta contenere insiemi e/o liste logiche.

Definizione 2.7 (Liste/Insiemi annidati e parzialmente specificati)**.** *Gli elementi di una lista logica (rispettivamente, un insieme logico) che sono a loro volta liste logiche (insiemi logici) si dicono* ***liste annidate (insiemi annidati)****.*
Una lista logica (insieme logico) che contiene almeno un oggetto logico unknown è detta ***parzialmente specificata****; se invece tutti gli oggetti della lista (insieme) sono known essa si dice* ***completamente specificata*** *o* ***ground****.*

Soffermiamoci ora sul valore relativo di un oggetto logico $X \hat{\in} \mathcal{L}$.
Supponiamo in particolare che $X \hat{\in} \mathcal{LV}$ e preso un certo $n \in N$ si abbia che $val^{(n)}(X) = v \hat{\notin} \mathcal{L}$. Si nota allora che il valore 'effettivo' di X non è $val(X)$ bensì v (potrebbe anche essere $v = \perp$).
Se invece $X \hat{\in} \mathcal{LS} \hat{\cup} \mathcal{LL}$, il ragionamento è simile ma il valore 'effettivo' non può essere $\perp$ (questo poichè mentre una variabile logica può assumere qualsiasi valore, una lista o un insieme sono comunque confinati ad essere una struttura dati ben precisa che contenga una collezione di elementi).
Definiamo quindi formalmente questo concetto di 'valore effettivo' o 'assoluto'.

Definizione 2.8 (Valore di un oggetto logico)**.** *Sia $X \hat{\in} \mathcal{L}$. Si definisce* ***valore (assoluto)*** *di X il valore $VAL(X)$ definito dalla funzione $VAL : \mathcal{L} \longmapsto \mathcal{O}$ tale che:*

$$VAL(X) = \begin{cases} val(X) & \text{se } val(X) \hat{\notin} \mathcal{L} \\ X & \text{se } (X \hat{\in} \mathcal{LS} \hat{\cup} \mathcal{LL}) \wedge (val(X) = \perp) \\ VAL(val(X)) & \text{altrimenti} \end{cases}$$

La funzione VAL(X) è implementata in JSetL dal metodo pubblico `X.getValue()`

Da una tale definizione si può ricavare qualche osservazione:

(i) Se $X \hat{\in} \mathcal{LV},\ \kappa(X) = 0$ sse $VAL(X) = \perp$

(ii) Se $X \hat{\in} \mathcal{LL} \hat{\cup} \mathcal{LS},\ \kappa(X) = 0$ sse $VAL(X) = X$

Per meglio comprendere questa definizione e le precedenti vengono riportati alcuni esempi:

Esempio 2.9. *Esempi di dichiarazione e utilizzo degli oggetti logici:*

1. `Lvar X = new Lvar();`
2. `Lvar Y = new Lvar("varY",'a');`
3. `Lvar Z = new Lvar(X);`
4. `Lst L = new Lst("L");`
5. `LSet S = new ConcreteLSet(2,5);`
6. `Lvar T = new Lvar(S);`
7. ```
   Object[] v = {1,X};
   Lst M = new Lst(v);
   ```
8. ```
   LSet[] w = {S,new ConcreteLSet("R1")};
   LSet R = new ConcreteLSet("R",w);
   ```

Vediamo ora di commentare queste istruzioni:

1. X è una var. logica non inizializzata e senza nome esterno:
   $X \hat{\in} \mathcal{LV},\ val(X) = VAL(X) = \perp,\ ext(X) = \perp,\ \kappa(X) = 0$
2. Y è una var. logica iniz. col carattere 'a' e nome esterno 'varY':
   $Y \hat{\in} \mathcal{LV},\ val(Y) = VAL(Y) = \mathtt{a},\ ext(Y) = \mathtt{varY},\ \kappa(X) = 1$
3. Z è una var. logica iniz. senza nome esterno che ha come valore relativo la var logica X:
   $Z \hat{\in} \mathcal{LV},\ val(Z) = X,\ VAL(Z) = \perp,\ ext(Z) = \perp,\ \kappa(Z) = 0$
4. L è una lista logica non inizializzata con nome esterno 'L':
   $L \hat{\in} \mathcal{LL},\ val(L) = \perp,\ VAL(L) = L,\ ext(L) = \mathtt{L},\ \kappa(Z) = 0$
5. S è un insieme logico iniz. col valore [2..5]={2,3,4,5} e senza nome esterno:
   $S \hat{\in} \mathcal{LS},\ val(S) = VAL(S) = \{2,3,4,5\},\ ext(S) = \perp,\ \kappa(Z) = 1$
6. T è una var. logica iniz. senza nome esterno che ha come valore relativo l'ins. logico S:
   $T \hat{\in} \mathcal{LV},\ val(T) = S,\ VAL(T) = \{2,3,4,5\},\ ext(T) = \perp,\ \kappa(T) = 1$

7. M è la lista parzialmente specificata ma limitata [1,X] (ha 2 elementi): $M \hat{\in} \mathcal{LL},\ val(M) = VAL(M) = [1, X],\ ext(M) = \perp,\ \kappa(M) = 1$

8. R = {{2,3,4,5},$R1$} è un insieme parzialmente specificato ma limitato (*al massimo* ha 2 elementi) mentre S ed $R1$ sono insiemi annidati (all'interno dell'insieme R): $R \hat{\in} \mathcal{LS},\ val(R) = VAL(R) = \{\{2, 3, 4, 5\}, R1\},\ ext(R) = \mathtt{R},\ \kappa(R) = 1$

Al contrario delle variabili logiche, le liste e gli insiemi possono essere manipolati anche attraverso appositi metodi di inserimento ed estrazione dei loro elementi. Tali metodi sono detti *costruttori di lista e di insieme* rispettivamente.

Definizione 2.10 (Costruttori di lista e di insieme)**.** *Siano* $e \hat{\in} \mathcal{O}$, $L \hat{\in} \mathcal{LL}$, $S \hat{\in} \mathcal{LS}$, $X \hat{\in} \mathcal{LV}$ *con* $\kappa(X) = 0$. *Un* ***costruttore di lista*** *è un espressione di una delle seguenti forme:*

(i) `L.ins1(e);`

(ii) `L.insn(e);`

(iii) `L.ext1(X);`

(iv) `L.extn(X);`

Un ***costruttore di insieme*** *è invece un espressione del tipo:*

(v) `S.ins(e);`

Le espressioni (i) e (ii) denotano la lista ottenuta aggiungendo ad L il valore di e, rispettivamente in testa o in coda alla lista.
Le espressioni (iii) e (iv) denotano la lista ottenuta rimuovendo rispettivamente il primo o l'ultimo elemento di L. Tale elemento diventerà quindi il valore di X, che quindi verrà inizializzata.
L'espressione (v) è equivalente all'insieme ottenuto 'aggiungendo' e ad S, cioè $S \cup \{e\}$.
Attraverso questi particolari costruttori, è possibile ottenere liste ed insiemi *illimitati*.

Esempio 2.11. *Esempi di utilizzo dei costruttori di lista e insieme:*

1. ```
   Lvar X = new Lvar();
   Lst L = Lst.empty.ins1(3+2).ins1(X);
   ```

2. ```
   Lvar Y = new Lvar("Y");
   Lst M = L.ext1(Y).insn(Y);
   ```

3.
```
LSet S = LSet.empty.ins(1).ins('a');
LSet R = new ConcreteLSet();
LSet T = R.ins(1);
```

Nell'esempio 1 costruisco la lista L partendo da [ ] e aggiungendo in testa 3+2=5 e la var. logica unknown X. In questo modo ottengo la lista L = [X,5] che è limitata e parzialmente specificata.
Nell'esempio 2 ottengo M dalla lista L: prima estraggo in Y il primo elemento di L (cioè X) e poi lo rimetto in coda. La lista M sarà quindi della forma [5,Y]≡[5,X] perchè VAL(Y)=X. Come L, M è limitata e parzialmente specificata (ma L≠M).
Nell'esempio 3 costruisco l'insieme S partendo da { } e aggiungendo gli elementi 1 e 'a', cioè S ={1,'a'}={'a',1}. Quindi, ottengo T dall'insieme *unknown* R attraverso l'inserimento dell'elemento 1: T = {1|R} (cioè, T={1} ∪ R).
Notiamo che S è limitato e completamente specificato, R non è specificato e T è illimitato (e di conseguenza parzialmente specificato).

2.2 La gestione dei vincoli

I vincoli in JSetL sono particolari condizioni su `Lvar` e `LSet` gestiti da un **Constraint Solver** (o risolutore di vincoli, di fatto è un istanza della classe di JSetL `SolverClass`) che ne implementa la strategia di risoluzione.
Tali vincoli sono passati al constraint solver S mediante l'inserimento in un **Constraint Store**, che contiene la collezione corrente Γ dei vincoli attivi in S.
La risoluzione di Γ è effettuata mediante la chiamata di uno dei **metodi di constraint solving** forniti dal solver, come ad esempio `solve()` e `finalSolve()`. Il solver ricerca (in modo non-deterministico) una *soluzione* che soddisfi tutti i vincoli di Γ, riscrivendoli se possibile in una **forma semplificata**.
Una tale trasformazione si basa su precise regole di semplificazione definite per ogni vincolo che occorre in Γ [1].
Formalmente, un vincolo è una particolare *relazione su un certo dominio* $\mathcal{D}$. In JSetL, il dominio dei vincoli è quello degli insiemi ($\mathcal{SET}$) ampliato con alcuni semplici vincoli sugli interi.

Definizione 2.12 (Vincolo atomico)**.** *Un* ***vincolo atomico*** *in JSetL è un'espressione di una delle seguenti forme:*

- `X.op();`

- `X.op(Y);`
- `X.op(Y,Z);`

dove `op` *è uno dei* ***metodi predefiniti*** *per la gestione dei vincoli (*`eq, neq, in, nin, subset, union, lt, le, gt, ge` *ecc...) e* `X,Y,Z` *sono espressioni che dipendono dal vincolo* `op` *(sicuramente, si avrà che* $\mathtt{X} \hat{\in} \mathcal{L}$*).*

Il significato di questi metodi è intuitivamente associato al loro nome: `eq` e `neq` sono vincoli di uguaglianza e disuguaglianza; `in` e `nin` sono vincoli di appartenenza e non appartenenza insiemistica; `subset` e `union` sono vincoli di inclusione e unione insiemistica; `lt, le, gt, ge` sono vincoli su interi corrispondenti rispettivamente alle relazioni $<, \leq, >, \geq$ e cosi via...

Definizione 2.13 (Vincolo)**.** *Un* ***vincolo*** *in JSetL è una congiunzione di uno o più vincoli atomici, cioè un'espressione della forma:*

- c_1`.and(`c_2`)....and(`c_n`)`

dove $c_1, c_2, \ldots, c_n$ *sono vincoli atomici e* $n \in N \setminus \{0\}$

Il significato di c_1`.and(`c_2`)....and(`c_n`)` è la **congiunzione logica** $\hat{c_1} \wedge \hat{c_2} \wedge \ldots \wedge \hat{c_n}$ dove $\hat{c_1}, \ldots, \hat{c_n}$ rappresentano il 'significato logico' delle espressioni $c_1, \ldots, c_n$.
Ad esempio, se c è il vincolo `X.subset(Y)`, allora $\hat{c}$ è $\mathrm{X} \subseteq \mathrm{Y}$; oppure, se c è il vincolo `a.nin(S)`, $\hat{c}$ è $\mathrm{a} \notin \mathrm{S}$ e così via...
D'ora in avanti, con un piccolo abuso di notazione, non si faranno differenze fra i c_i ed i $\hat{c_i}$ che saranno utilizzati in modo equivalente a seconda del contesto. Inoltre, potrà essere utilizzata anche una *notazione prefissa*: se `c` è un vincolo della forma:

- `X.op();` lo indicheremo anche con $op(X)$
- `X.op(Y);` lo indicheremo anche con $op(X, Y)$
- `X.op(Y,Z);` lo indicheremo anche con $op(Y, Z, X)$

Ad esempio, il vincolo S3.`union`(S1,S2) potrà anche essere indicato come $\mathrm{S3} = \mathrm{S1} \cup \mathrm{S2}$ oppure *union*(S1,S2,S3).
Vediamo ora alcuni esempi di vincoli in JSetL:

Esempio 2.14. *Siano* `X,Y,Z` $\hat{\in} \mathcal{LV}$*;* `R,S,T` $\hat{\in} \mathcal{LS}$ *e* `N` $\hat{\in} \mathcal{I}$

1. `R.eq(S);`
2. `T.union(R,S);`

3. `X.neq(Y.sum(N).and(X.eq(3)).and(Y.neq(Z));`

4. `Z.ge(-6);`

Il vincolo 1 rappresenta l'ugugaglianza insiemistica `R=S`. Si noti che ciò è ben diverso da un assegnamento ed implica il concetto di **unificazione** tra R ed S. Questo significa che il vincolo atomico R=S impone la ricerca di un assegnamento agli elementi di R ed S tale che i due insiemi risultino uguali secondo la teoria degli insiemi.
In generale, un problema di unificazione può ammettere più di una soluzione: ad esempio, il vincolo {X,Y}={Z,2} con X,Y,Z variabili logiche unknown risulta soddisfatto con X=Z $\wedge$ Y=2 ma anche con Y=Z $\wedge$ X=2. La gestione di queste soluzione multiple è affidato, come si vedrà in seguito, al constraint solver.
L'esempio 2 esprime il vincolo atomico *union*(R,S,T) che è soddisfatto se $T = R \cup S$.
L'esempio 3 rappresenta il vincolo $X \neq (Y + N) \wedge X = 3 \wedge Y \neq Z$.
L'esempio 4 corrisponde al vincolo atomico $Z \geq -6$.

2.2.1 Constraint store e constraint solving

Il **constraint store** (o più sinteticamente *c.store*) di un constraint solver S contiene la collezione di tutti i vincoli attualmente attivi in S. JSetL fornisce metodi attraverso i quali è possibile aggiungere nuovi vincoli al constraint store, visualizzarne il contenuto, rimuovere tutti i vincoli in esso presenti.
Se il c.store contiene n vincoli (atomici e non) $c_1, c_2, \ldots, c_n$ allora indicheremo con Γ il vincolo corrispondente alla congiunzione dei vincoli $c_1, \ldots, c_n$, cioè $\Gamma = c_1 \wedge c_2 \wedge \ldots \wedge c_n$.
L'**inserimento** di un nuovo vincolo `C` al c.store è effettuato con la chiamata al metodo `add` della classe `SolverClass`: l'invocazione `S.add(C)` comporta l'aggiunta di `C` al c.store del contraint solver `S`. Il significato di tale istruzione è l'aggiornamento del vincolo Γ, cioè $\Gamma \leftarrow \Gamma \wedge$ `C`.
Una volta aggiunti nuovi vincoli al c.store, si può chiedere al solver S di *risolverli* invocando uno degli appositi metodi. Il **constraint solving** (o più sinteticamente *c.solving*) in JSetL è basato sulla riduzione di ogni vincolo atomico in una forma semplificata, chiamata *solved form*, che è dimostrato essere **soddisfacibile**. Il successo di questa riduzione su tutti i vincoli atomici di Γ permette quindi di concludere che la collezione dei vincoli del c.store è sicuramente **corretta**. In caso contrario, l'individuazione di un *fallimento* (in termini logici, una riduzione a `false`) implica la non-soddisfacibilità di Γ.

Definizione 2.15 (Solved Form)**.** *Siano dati:*

- $X \hat{\in} \mathcal{L}$ *con* $\kappa(X) = 0$
- $Z, Z_1, Z_2, Z_3 \hat{\in} \mathcal{LV} \hat{\cup} \mathcal{LS}$ *con* $\kappa(Z) = \kappa(Z_1) = \kappa(Z_2) = \kappa(Z_3) = 0$
- $E \hat{\in} \mathcal{L}$

Un vincolo C è in ***solved form*** *se è vuoto oppure tutti i vincoli atomici che lo compongono hanno una delle seguenti forme:*

1. $X = E$,
   e X non occorre in E o in altri vincoli di C
2. $X \neq E$,
   e X non occorre in E
3. $E \notin Z$,
   e Z non occorre in E
4. $disj(Z_1, Z_2)$,
   e $Z_1 \not\equiv Z_2$
5. $union(Z_1, Z_2, Z_3)$,
   con $Z_1 \not\equiv Z_2$ *e non esistono vincoli della forma* $Z_i \neq E$, *con* $i = 1, 2, 3$, *all'interno del vincolo C.*

Una solved form ottenuta da un vincolo C rappresenta una **soluzione** per C. Un vincolo C può avere più di una soluzione: in questo caso il c.solver di JSetL cerca tutte le soluzioni di C in modo *non-deterministico* utilizzando strumenti come punti di scelta e backtracking (vedi prossima sezione). Tuttavia, è anche possibile che C non abbia soluzione: in questo caso il processo di riduzione fallisce e viene sollevata l'eccezione `Failure`.
Diremo che una computazione **termina con fallimento** se essa causa la generazione dell'eccezione `Failure`; in caso contrario si dirà che la computazione **termina con successo**.
I vincoli in solved form sono *irriducibili* mentre quelli non in solved form non vengono mantenuti nel c.store: questi infatti o vengono messi in solved form o vengono riscritti a `false` oppure causano la generazione di un eccezione.
L'approccio al constraint solving di JSetL è fondamentalmente quello di CLP($\mathcal{SET}$), esteso con **semplici vincoli sugli interi**.
Tuttavia, il solver di JSetL non è in grado di trattare questi vincoli in modo esauriente: se nel processo di risoluzione vengono individuati vincoli su interi contenenti variabili logiche non inizializzate, viene sollevata l'eccezione

`Unitialized_Variable_in_arithmetical_expression`.
Un modo per aggirare questo problema è l'utilizzo di una forma risolta più '*debole*'.

Definizione 2.16 (Weak Solved Form)**.** *Siano dati:*

- $X \hat{\in} \mathcal{L}$ *con* $\kappa(X) = 0$
- $Z, Z_1, Z_2, Z_3 \hat{\in} \mathcal{LV} \hat{\cup} \mathcal{LS}$ *con* $\kappa(Z) = \kappa(Z_1) = \kappa(Z_2) = \kappa(Z_3) = 0$
- $E \hat{\in} \mathcal{L}$
- $A_1, A_2 \hat{\in} \mathcal{LV}$ *e* $A_3 \hat{\in} LV \hat{\cup} \mathcal{I}$
- $op' \in \{+, -, *, /, mod\}$
- $op'' \in \{<, \leq, \geq, >\}$

Un vincolo C è in ***weak solved form*** *se è vuoto oppure tutti i vincoli atomici che lo compongono hanno una delle seguenti forme:*

1. $X = E$,
   e X *non occorre in* E *o in altri vincoli di* C
2. $X \neq E$,
   e X *non occorre in* E
3. $E \notin Z$,
   e Z *non occorre in* E
4. $disj(Z_1, Z_2)$,
   e $Z_1 \not\equiv Z_2$
5. $union(Z_1, Z_2, Z_3)$,
   e $Z_1 \not\equiv Z_2$
6. $A_1 = A_2\ op'\ A_3$,
   con $\kappa(A_2) = 0 \vee \kappa(A_3) = 0$ *e* $\kappa(A_i) = 1 \rightarrow A_i \hat{\in} \mathcal{I}$ *per* $i = 1, 2, 3$
7. $A_1\ op''\ A_3$,
   con $\kappa(A_1) = 0 \vee \kappa(A_3) = 0$ *e* $\kappa(A_i) = 1 \rightarrow A_i \hat{\in} \mathcal{I}$ *per* $i = 1, 2$

La prima cosa che si nota è che le prime 4 condizioni della weak solved formed **coincidono esattamente** con quelle della solved form. La condizione 5 è molto simile, ma le condizioni sulla *union* della solved form sono **rilassate**: infatti, evito di imporre la non-esistenza di vincoli della forma

$Z_i \neq E$ per $i = 1, 2, 3$ all'interno di C. Inoltre, vengono aggiunte le condizioni 6 e 7 che permettono di non sollevare eccezioni qualora si provi a risolvere vincoli su interi contenenti variabili logiche non inizializzate.
La weak solved form è sicuramente più comoda ed efficiente della solved form, ma ha una rilevante limitazione: una weak solved form ottenuta da un vincolo C **non garantisce** che C abbia una *soluzione*. Si considerino infatti i seguenti esempi:

Esempio 2.17 (Weak solved form non soddisfacibili)**.** *Siano dati i seguenti vincoli:*

(c1) `X.lt(Y).and(X.gt(Y));` *con* `X,Y` $\hat{\in} \mathcal{LV}$ *e* κ*(*`X`*)=*κ*(*`Y`*)=0*

(c2) `S.subset(T).and(T.subset(S)).and(S.neq(T));` *con* `S,T` $\hat{\in} \mathcal{LS}$ *e* κ*(*`S`*)=*κ*(*`T`*)=0*

Allora **c1** *e* **c2** *sono vincoli riducibili in forma risolta debole, ma chiaramente non sono soddisfacibili (si dice anche che sono* ***inconsistenti****).*
Se provassi a ridurli in forma risolta, avrei una terminazione della computazione:

- **c1** *provoca l'eccezione* `Unitialized_Variable_in_arithmetical_expression`
- **c2** *provoca l'eccezione* `Failure` *(vìolo la condizione 5 della solved form)*

Vediamo ora quali sono i metodi di c.solving che posso invocare attraverso oggetti della classe `SolverClass`.

Definizione 2.18 (`solve` e `finalSolve`)**.** *Siano S il contraint solver (ovvero, un'istanza della classe* `SolverClass`*),* Γ *il vincolo corrente del c.store di S e C un vincolo qualsiasi di JSetL.*
L'invocazione del metodo S.`solve()` *(rispettivamente, S.*`solve`*(C)) termina con* ***successo*** *se* Γ *(rispettivamente,* $\Gamma \wedge C$*) può essere ridotto in weak solved form; termina con* ***fallimento*** *altrimenti.*
L'invocazione del metodo `S.finalSolve()` *(rispettivamente,* `S.finalSolve(C)`*) termina con successo se* Γ *(rispettivamente,* $\Gamma \wedge C$*) può essere ridotto in solved form; termina con fallimento altrimenti.*

D'ora in avanti indicheremo con WSF la weak solved form e con SF la solved form. Si noti che se un vincolo è in WSF e non contiene confronti tra variabili logiche non inizializzate oppure vincoli di unione, allora tale vincolo è anche in SF (quindi soddisfacibile). Tuttavia in generale per ottenere la

sicurezza della SF bisogna invocare il metodo `finalSolve`.
I vincoli possono essere aggiunti al c.store e risolti tutti insieme (l'ordine di inserimento è assolutamente ininfluente) oppure è possibile risolvere uno ad uno i vincoli immediatamente dopo l'inserimento: il risultato è equivalente.
Esistono anche altri metodi di c.solving quali ad esempio `boolSolve`, `solve1`, `nextSolution`, `setof`... ma per il momento non ce ne occuperemo.

2.3 Il non-determinismo

Una computazione in JSetL può essere **non-deterministica**, sebbene il concetto di non-determinismo in JSetL è confinato al c.solving e più precisamente alle operazioni su insiemi (come in Singleton).
Un classico esempio di operazione non-deterministica è l'*unificazione insiemistica*. Si consideri infatti il problema di unificazione $\{X, Y, Z\} = \{1, 2, 3\}$; il c.solving ritorna, in modo non-deterministico, sei differenti soluzioni (che sono di fatto tutte le permutazioni dell'insieme $\{1, 2, 3\}$) cioè:

$$\begin{array}{lll} X = 1 & Y = 2 & Z = 3 \\ X = 1 & Y = 3 & Z = 2 \\ & \cdots & \\ X = 3 & Y = 2 & Z = 1 \end{array}$$

Un altro esempio è il problema di appartenenza insiemistica $X \in \{1, 2, 3\}$, con $X \hat{\in} \mathcal{LV}$ e $\kappa(X) = 0$, che ritorna in modo non deterministico le soluzioni

$$X = 1 \vee X = 2 \vee X = 3$$

Il non-determinismo fornisce un livello di astrazione molto alto e permette un forte supporto alla programmazione dichiarativa.
JSetL offre più di un metodo per la gestione del non-determinismo; siano infatti `S` un oggetto di classe `SolverClass`, `X` una variabile logica e `C` un vincolo, si considerino i seguenti metodi:

- `S.nextSolution()`: Permette di ottenere, se esiste, una nuova soluzione dell'ultimo problema risolto; se non vi sono altre soluzioni ritorna `false`
- `S.setof(X,C)` (rispettivamente, `S.setof(X)`): ritorna un insieme costituito da tutti i valori di X che soddisfano il vincolo $\Gamma \wedge$ C (rispettivamente, Γ).
- `S.solve1()`: funziona come la `solve` ma tiene conto solamente della prima soluzione, ignorando tutte le possibili altre alternative non-deterministiche.

Un potente strumento offerto da JSetL è la possibilità di implementare procedure non-deterministiche. Ciò è possibile implementando uno speciale costrutto astratto ***either-orelse*** che permette di esprimere una scelta non-deterministica tra due o più statements. Siano infatti $S_1, ..., S_n$ statements di Java, il costrutto

$$\textbf{\textit{either}}\ S_1\ \textbf{\textit{orelse}}\ S_2\ \ldots\ \textbf{\textit{orelse}}\ S_n$$

ha come significato logico la disgiunzione $S_1 \vee S_2 \vee ... \vee S_n$.
L'interpretazione computazionale di questo costrutto è l'esplorazione, attraverso il backtracking, di tutte le possibili alternative $S_1, ..., S_n$ a partire da S_1.
L'implementazione in codice Java è lasciata alla prossima sezione, per ora ci si limiti ad un esempio in pseudo-codice:

Esempio 2.19 (costrutto *either-orelse*)**.** *Siano* $L_1, L_2, L_3 \hat{\in} \mathcal{LL}$*, il vincolo*

$$\textbf{\textit{concat}}(L_1, L_2, L_3)$$

è soddisfatto se L_3 *è la concatenazione delle liste* L_1 *ed* L_2*.*
La seguente funzione implementa tale vincolo utilizzando il costrutto astratto ***either-orelse****.*

```
public void concat(Lst L1, Lst L2, Lst L3) {
  either { // Statement S1
   Solver.add(L1.eq(Lst.empty)); // L1 = []
   Solver.add(L2.eq(L3)); // L2 = L3
  }
  orelse { // Statement S2
   Lvar X = new Lvar();
   Lst R1 = new Lst();
   Lst R3 = new Lst();
   Solver.add(L1.eq(R1.ins1(X))); // L1 = [X|R1]
   Solver.add(L3.eq(R3.ins1(X))); // L3 = [X|R3]
   Solver.add(concat(R1,L2,R3));
  }
  return;
}
```

Il metodo `concat` implementa la disgiunzione logica degli statement S1 $\vee$ S2. In pratica, nello statement S1 si ha che se L1 è la lista vuota allora la concatenazione di L1 ed L2 è proprio L2.
In S2 invece L1 ed L3 hanno il primo elemento X in comune: devo imporre

che R3 (cioè il resto della lista L3) sia la concatenazione di R1 (resto di L1) ed L2.
Si noti che per implementare S2 faccio una *chiamata ricorsiva* di `concat`: questo tipo di approccio è molto frequente nella programmazione logica.
Conoscendo le basi del linguaggio logico *Prolog*, si osserva immediatamente che la semantica di questa funzione coincide con quella delle seguenti *clausole di Horn*:

```
concat([],L2,L2).
concat([X|R1],L2,[X|R3]):-
   concat(R1,L2,R3).
```

2.4 Definizione di nuovi vincoli

JSetL permette all'utente di definire nuovi vincoli (detti anche *esterni*), trattati allo stesso modo di quelli built-in (*interni*): sarà quindi possibile aggiungerli al c.store e risolverli utilizzando gli appositi metodi di `SolverClass`.
La definizione di nuovi vincoli è fornita mediante l'implementazione di una classe definita da utente che estenda la classe astratta di JSetL `NewConstraintClass`, rispettando opportune convenzioni di programmazione.
Ad esempio, la funzione dell'esempio 2.19 può essere definita come vincolo esterno: basta costruire una classe che erediti da `NewConstraintClass` e implementare gli appositi metodi. Vediamo come è possibile realizzare tutto ciò:

Esempio 2.20. (vincolo `concat` definito da utente)

```
import JSetL.*;

public class MyConstraints extends NewConstraintsClass {

  public MyConstraints(SolverClass CurrentSolver) {
    super(CurrentSolver);
  }

  public Constraint concat(Lst L1, Lst L2, Lst L3) {
    return new Constraint("concat",L1,L2,L3);
  }

  public void user_code(Constraint c)
```

```
    throws Failure, NotDefinedConstraint {
      if(c.getName()=="concat")
       concat(c);
      else
        throw new NotDefinedConstraint();
      return;
    }

    public void concat(Constraint c)
    throws Failure {
      Lst L1 = (Lst)c.getArg(1);
      Lst L2 = (Lst)c.getArg(2);
      Lst L3 = (Lst)c.getArg(3);
      switch(c.getAlternative()) {
        case 0:
          Solver.addChoicePoint(c);
          Solver.add(L1.eq(Lst.empty)); // L1 = []
          Solver.add(L2.eq(L3)); // L2 = L3
          break;
        case 1:
          Lst X = new Lst();
          Lst R1 = new Lst();
          Lst R3 = new Lst();
          Solver.add(L1.eq(R1.ins1(X))); // L1 = [X|R1]
          Solver.add(L3.eq(R3.ins1(X))); // L3 = [X|R3]
          Solver.add(concat(R1,L2,R3));
          break;
      }
      return;
    }
  }
```

La classe definita per il nuovo vincolo ha solamente un costruttore (con un solo parametro di tipo `SolverClass`), che non fa altro che richiamare l'analogo costruttore della classe da cui eredita (`NewConstraintsClass`). Il parametro `CurrentSolver` si riferisce al c.solver attualmente in uso dall'utente.
Il metodo `concat` ritorna un oggetto di classe `Constraint`, ovvero una struttura dati che implementa un qualsiasi vincolo atomico di JSetL: il costruttore

con 4 parametri utilizzato crea quindi un vincolo di nome '`concat`' avente come parametri le liste `L1`, `L2` ed `L3`.
Il metodo `user_code`, definito nella classe astratta `NewConstraintClass`, associa ad ogni nome di vincolo definito da utente il corrispondente metodo che lo implementa. In questo modo, all'interno della classe `MyConstraints` potrei definire più di un vincolo, discriminato univocamente in base al suo nome. Se il metodo `user_code` viene richiamato su di un vincolo non implementato, allora viene sollevata l'eccezione `NotDefinedConstraint`.
Infine, all'interno del metodo `concat` si trova la definizione vera e propria del vincolo. Da notare che, al contrario dell'esempio 2.19, il costrutto ***either-orelse*** è implementato interamente in codice Java. In pratica, se devo implementare la scelta non-deterministica $S_1 \vee S_2 \vee \ldots \vee S_n$ con $S_1, \ldots, S_n$ statement Java, mi basta costruire uno statement di tipo `switch` come il seguente:

```
switch( C.getAlternative() ) {

  case 0:
    S.addChoicePoint(C);
      // codice dello statement S_1
  case 1:
    S.addChoicePoint(C);
      // codice dello statement S_2
  ...

  case n-2:
    S.addChoicePoint(C);
      // codice dello statement S_{n-1}
  case n-1:
      // codice dello statement S_n
}
```

dove `C` è il vincolo che devo implementare e `S` il c.solver corrente.
Il metodo `C.getAlternative()` permette di utilizzare le informazioni di controllo contenute in `C`, in modo da poter gestire le n possibili alternative. Ogni blocco `case`, eccetto l'ultimo, crea un *punto di scelta* e lo aggiunge allo *stack delle alternative* grazie all'invocazione del metodo S.addChoicePoint(C).
In questo modo, al termine di un qualsiasi statement S_i, attraverso la tecnica del *backtracking* posso risalire all'ultimo punto di scelta lasciato aperto (se esiste) e nel caso esplorare una delle altre alternative.

Capitolo 3

Il vincolo di cardinalità insiemistica

In matematica, presi due insiemi A e B essi si dicono *equipotenti* se esiste una biiezione f:$A \longmapsto B$.
Tale relazione di equipotenza è ovviamente una relazione di *equivalenza*: se due insiemi appartengono alla stessa classe di equivalenza (cioè, sono equipotenti) allora si dice che hanno la stessa **cardinalità**. D'ora in avanti, se S è un insieme indicheremo con $\#S$ la sua cardinalità.
Nel caso di insiemi *finiti*, la cardinalità sarà quindi un numero naturale, positivo o nullo, corrispondente al numero di elementi dell'insieme.
In informatica, il concetto di cardinalità è rilevante soprattutto per quanto concerne la *teoria della calcolabilità* [8]; tuttavia, per i nostri scopi in JSetL considereremo la cardinalità (o *size*) solamente come un particolare *vincolo su insiemi finiti*.
Formalmente, un tale vincolo sarà una relazione sul dominio $\mathcal{D} = \mathcal{LS} \hat{\cup} \mathcal{LV}$ (come visto nei paragrafi precedenti, una variabile logica può assumere un qualsiasi tipo di valore). Si osservi che tale relazione è in particolare una **funzione** che ha come dominio $\mathcal{D}$ e come codominio $I \hat{\cup} LV$, cioè

$$size : \mathcal{LS} \hat{\cup} \mathcal{LV} \longmapsto I \hat{\cup} LV$$

Nella prossima sezione definiremo quindi il vincolo *size* e la sua risoluzione.

3.1 Definizione e risoluzione del vincolo *size*

Definizione 3.1 (Vincolo di cardinalità insiemistica)**.** *Il vincolo di cardinalità insiemistica in JSetL è definito come* ***size(S,N)*** *ed è soddisfatto se e solo se sono soddisfatte le seguenti tre condizioni:*

(i) $(S \hat{\in} \mathcal{LS}) \vee (S \hat{\in} \mathcal{LV} \wedge VAL(S) \hat{\in} \mathcal{LS}) \vee (S \hat{\in} \mathcal{LV} \wedge \kappa(S) = 0)$

(ii) $(N \hat{\in} \mathcal{I}) \vee (N \hat{\in} \mathcal{LV} \wedge VAL(N) \hat{\in} \mathcal{I}) \vee (N \hat{\in} \mathcal{LV} \wedge \kappa(N) = 0)$

(iii) $\#S = N$

Questo significa che S può essere un insieme qualsiasi, un `Lvar` instanziata su un insieme oppure un `Lvar` unknown. Analogamente, N può essere un intero qualsiasi, un `Lvar` instanziata su un intero oppure un `Lvar` unknown. Questi due oggetti sono vincolati dal fatto che la cardinalità di S dev'essere N.
Ovviamente, conoscere la cardinalità di un insieme *ground* è piuttosto banale; sia infatti $S = \{1, -3, 9.2\}$ un insieme completamente specificato, si può tranquillamente affermare che $\#S = 3$.
E se invece fosse $S = \{X, Y\}$ con X e Y oggetti logici unknown?
A prima vista verrebbe da dire che $\#S = 2$. Tuttavia ciò è vero solo nel caso in cui $X \neq Y$; se invece fosse $X = Y$ allora si avrebbe $\#S = 1$ in quanto $S = \{X\} = \{Y\}$.
Un altro esempio è $S = \{a|R\}$: se $a \notin R$, allora $\#S = \#R + 1$, ma se $a \in R$ si avrà che $\#S = \#R$.

La gestione di questi casi (e molti altri) dev'essere sicuramente *non-deterministica*; per il momento però non ci occuperemo dell'implementazione vera e propria del vincolo *size* in codice Java (rimandata al capitolo 7) ma verranno effettuate considerazioni a livello più astratto.

Definizione 3.2 (Regole di risoluzione del vincolo *size*)**.** *Sia $size(S, N)$ un vincolo che rispetta le condizioni della definizione 3.1, allora:*

(r1) *Se S è un insieme **ground** della forma $\{a_1, a_2, \ldots, a_k\}$ con tutti gli elementi a_i specificati, allora si avrà sicuramente che $\#S = k$: devo solamente sostituire $size(S, N)$ col vincolo atomico $N = k$ (in particolare, se $S = \{\}$ si avrà che $N = 0$).*

(r2) *Se $\kappa(S) = \kappa(N) = 0$, allora tale vincolo si dirà in **forma irriducibile**, in quanto non è possibile ricavare nessun'altra informazione dai suoi parametri: esso verrà mantenuto nel c.store fino a che non verranno adeguatamente instanziati S oppure N.*

(r3) *Se $\kappa(S) = 0$ e $\kappa(N) = 1$ allora preso $k = \begin{cases} N & \text{se } N \hat{\in} \mathcal{I} \\ VAL(N) & \text{se } N \hat{\in} \mathcal{LV} \end{cases}$*

*devo imporre che S abbia esattamente **k elementi distinti** tra loro, ossia:*

- $S = \{X_1, X_2, \ldots, X_k\}$ *con* $X_i \hat{\in} \mathcal{LV}$ *e* $\kappa(X_i) = 0$ *per* $i = 1, \ldots, k$
- $X_i \neq X_j$ *per* $1 \leq i < j \leq k$

In particolare, se $k = 0$ *si avrà che* $S = \{\}$

(r4) *Per tutti gli altri casi, devo utilizzare una definizione* ***non-deterministica****:*

either $\{$
$S = \{e|R\} \wedge N \geq 1 \wedge e \notin R \wedge M = N - 1 \wedge \ size(R, M)$
$\}$
orelse $\{$
$S = \{e|R\} \wedge N \geq 1 \wedge R = \{e|R1\} \wedge e \notin R1 \wedge M = N - 1 \wedge \ size(R1, M)$
$\}$

Si osservi che nei primi tre casi, pur non essendo specificata l'implementazione, la soluzione del vincolo è puramente *deterministica* ed il costo sarà sicuramente polinomiale:

(r1) Sostituisco $size(S, N)$ col vincolo di uguaglianza $N = k$.

(r2) Mantengo il vincolo $size(S, N)$ nel c.store senza effettuare alcuna operazione.

(r3) Sostituisco $size(S, N)$ con un vincolo di uguaglianza insiemistica $S = \{X_1, \ldots, X_k\}$ e $\sum_{i=1}^{k-1} i = \frac{k(k-1)}{2} = \frac{k^2}{2} - \frac{k}{2}$ vincoli di disuguaglianza della forma $X_i = X_j$.

Viceversa, la regola **(r4)** (che gestisce tutte le altre possibili situazioni) invoca l'utilizzo del non-determinismo: ciò comporta in generale una perdita di efficienza della computazione.

3.2 Esempi di risoluzione

Vediamo ora qualche esempio pratico di risoluzione del vincolo $size(S, N)$ attraverso semplici esempi.
Siano α, β vincoli di JSetL; se β è il vincolo ottenuto dalla risoluzione dei vincoli *size* che occorrono in α, allora scriveremo che $\alpha \Rightarrow \beta$. Si noti che l'operatore $\Rightarrow$ non è funzionale, in quanto il non-determinismo della risoluzione potrebbe determinare k soluzioni distinte $\beta_1, \ldots, \beta_k$: in questo caso allora $\alpha \Rightarrow \beta_1 \vee \ldots \vee \alpha \Rightarrow \beta_k$. In particolare, potrebbe non esistere alcuna soluzione: scriveremo allora $\alpha \Rightarrow$ `false`.
Consideriamo quindi i seguenti esempi:

Esempi 3.3 (Risoluzione del vincolo *size*)**.**

1. $size(\{\}, N)$ con $\kappa(N) = 0$ (r1)
   $\Rightarrow N = 0$

2. $size(S, 0)$ con $\kappa(S) = 0$ (r3)
   $\Rightarrow S = \{\}$

3. $size([a..b], N)$ con $\kappa(N) = 0$ e $a, b \,\hat{\in}\, \mathcal{I}$ (r1)
   $\Rightarrow N = b - a + 1$

4. $size(S, -3)$ con $\kappa(S) = 0$ (r3)
   $\Rightarrow$ `false`

5. $size(\{a, b, c, b\}, N)$ con $\kappa(N) = 0$, $a \neq b \neq c$ e $a, b, c \,\hat{\notin}\, \mathcal{L}$ (r1)
   $\Rightarrow N = 3$

6. $size(\{X, Y\}, N)$ con $\kappa(X) = \kappa(Y) = \kappa(N) = 0$ (r4)
   $\Rightarrow X \neq Y \ \wedge \ N = 2$
   $\Rightarrow X = Y \ \wedge \ N = 1$

7. $size(S, 2)$ con $\kappa(S) = 0$ (r3)
   $\Rightarrow S = \{X_1, X_2\} \ \wedge \ X_1 \neq X_2$

8. $size(\{1, 3, 2\}, 5)$ (r1)
   $\Rightarrow$ `false`

9. $size(\{a|R\}, 2)$ con $\kappa(R) = 0$ e $a \,\hat{\notin}\, \mathcal{L}$ (r4)
   $\Rightarrow R = \{X\} \ \wedge \ X \neq a$
   $\Rightarrow R = \{a, X\} \ \wedge \ X \neq a$

10. $size(\{a|R\}, N)$ con $\kappa(R) = \kappa(N) = 0$ e $a \,\hat{\notin}\, \mathcal{L}$ (r4)
    $\Rightarrow a \notin R \ \wedge \ size(R, M) \ \wedge \ N \geq 1 \ \wedge \ M = N - 1$
    $\Rightarrow R = \{a|R_1\} \ \wedge \ a \notin R_1 \ \wedge \ size(R_1, M) \ \wedge \ N \geq 1 \ \wedge \ M = N - 1$

11. $size(S, N)$ con $\kappa(S) = \kappa(N) = 0$ (r2)
    $\Rightarrow size(S, N)$

12. $size(\{\{1, 2\}, \{2, 1\}\}, N)$ con $\kappa(N) = 0$
    $\Rightarrow N = 1$

Accanto ad ogni vincolo è indicata la regola utilizzata per risolverlo, mentre nella riga sottostante è presente la sua risoluzione (o le sue risoluzioni, come nei casi non-deterministici 6-9-10).

Si noti che negli esempi 4 e 8 ottengo **false** in quanto vìolo la regola (iii) della definizione 3.1: nell'esempio 4 ciò avviene perchè $\#S = -3 < 0$, mentre nell'8 si ha che $\#S = 3 \neq 5$.
Negli altri esempi, la risoluzione è deterministica e piuttosto intuitiva.

D'ora in avanti quindi quando parleremo di SF e WSF intenderemo forme risolte **'estese' al vincolo size**, cioè le definizioni 2.15 e 2.16 dovranno includere il caso in cui un vincolo in SF o WSF possa contenere anche un vincolo atomico della forma $size(S, N)$ con $\kappa(S) = \kappa(N) = 0$.
Purtroppo, si noterà come questa estensione causerà la possibile **perdita della soddisfacibilità globale** del c.store.
In altre parole, l'inserimento di $size(S, N)$ porterà ad avere situazioni in cui un vincolo non corretto non sia riconosciuto come tale dal c.solver.
Si osservi inoltre che la risoluzione (r4) risulta spesso in collisione con la riduzione in SF, in quanto tra i vincoli trattati vi sono $N \geq 1$ e $M = N - 1$ che, se $\kappa(N) = 0$, causano il *fallimento* della computazione.

3.3 Proprietà della cardinalità

Nel precedente paragrafo si è potuto notare come la risoluzione del vincolo *size* avviene '**localmente**', cioè risolvendo indipendentente ogni singolo vincolo senza tenere conto delle possibili interferenze che esso può avere con altri vincoli atomici. Consideriamo allora i seguenti vincoli in WSF:

- $size(S, N) \ \wedge \ N < 0$
- $union(X, Y, Z) \ \wedge \ size(X, N) \ \wedge \ size(Z, M) \ \wedge \ N > M$

Si nota immediatamente come questi vincoli siano chiaramente inconsistenti, infatti:

- La cardinalità di un insieme finito non può essere un numero negativo
- Se $Z = X \cup Y$, allora $\#X \leq \#Z$

Quindi, le sole regole 3.2 non permettono in generale di ottenere un vincolo consistente. Perciò, se nel c.store sono presenti vincoli di *size* in forma irriducibile non è garantita la soddisfacibilità dello stesso: bisognerebbe disporre di adeguati **controlli globali** su tutti i vincoli atomici di Γ.
Questo approccio verrà trattato con cura nel capitolo 5, per ora ci si limiti ad osservare come il vincolo *size* goda di proprietà che lo legano sia al dominio degli insiemi che a quello degli interi. Ciò, come vedremo in seguito, sarà la

principale fonte dei problemi riguardanti la gestione completa del vincolo.
In questa sezione verranno quindi presentate alcune proprietà caratteristiche del vincolo *size* che non vengono gestite correttamente dall'attuale solver di JSetL nel caso in cui esso si presenti in **forma irriducibile**.
Infatti, se $\kappa(S) = 1 \vee \kappa(N) = 1$, si può notare come le regole (r1), (r3) ed (r4) sostituiscano $size(S, N)$ con altri vincoli atomici che il solver riesce a ridurre quantomeno in WSF (uguaglianze, disuguaglianze, confronti fra interi...).

3.3.1 Cardinalità e operazioni fra interi

All'inizio del capitolo si è data la definizione di cardinalità di un insieme finito la quale, in parole povere, corrisponde al numero di elementi di tale insieme. Ovviamente, questo numero non potrà mai essere negativo: se ho un vincolo $size(S, N)$ allora deve per forza essere che $N \geq 0$. Ma questa non è l'unica proprietà che deve soddisfare il parametro N: ad esempio, se $size(S, N) \wedge N \neq 0$ allora $S \neq \{\}$; se $size(S, N) \wedge N \geq k$ allora S deve avere almeno k elementi e così via...
Vediamo quindi di riassumere tali relazioni in un quadro più generale.

Proprietà 3.4. *Si consideri un vincolo di JSetL del tipo*

$$size(S, N) \ \wedge \ N \ op \ M$$

con $op \in \{\neq, <, \leq, \geq, >\}$ *e* $M \hat{\in} \mathcal{LV} \hat{\cup} \mathcal{I}$ *tale che* $M \hat{\in} \mathcal{LV} \rightarrow VAL(M) \hat{\in} \mathcal{I}$.

$$Se\ k = \begin{cases} M & se\ M \hat{\in} \mathcal{I} \\ VAL(M) & se\ M \hat{\in} \mathcal{LV} \end{cases}$$

allora valgono le seguenti proprietà:

- *Se* $N \neq k$, *allora* ***either*** $N < k$ ***orelse*** $N > k$
- *Se* $N < k$, *con* $k > 0$, *allora:*
  either $size(S, 0)$ ***orelse*** $size(S, 1)$ ***orelse*** $\ldots$ ***orelse*** $size(S, k-1)$
- *Se* $N \leq k$, *con* $k \geq 0$, *allora:*
  either $size(S, 0)$ ***orelse*** $size(S, 1)$ ***orelse*** $\ldots$ ***orelse*** $size(S, k)$
- *Se* $N > k$, *con* $k \geq 0$, *allora:*
  $S = \{X_1, \ldots, X_k, X_{k+1} | R\} \ \wedge \ \bigwedge_{0 \leq i < j \leq k+1} (X_i \neq X_j) \ \wedge \ size(R, L) \wedge N = L + k + 1$
- *Se* $N \geq k$, *con* $k \geq 0$, *allora:*
  $S = \{X_1, \ldots, X_k | R\} \wedge \bigwedge_{0 \leq i < j \leq k+1} (X_i \neq X_j) \wedge size(R, L) \wedge N = L + k$

Purtroppo, queste deduzioni non sono naturalmente implementate in JSetL: bisognerà quindi definire una strategia globale che ne imponga rispetto. Ovviamente, tali considerazioni ad 'alto livello' non verranno poi completamente tradotte in codice; alcuni di questi casi potrebbero infatti causare un 'esplosione' di vincoli con conseguente perdita (anche notevole) di efficienza. Ma questo per ora non importa, occupiamoci invece di un'altra importante proprietà caratteristica del vincolo di cardinalità.

Come accennato nell'introduzione del capitolo 3, la relazione di cardinalità è in particolar modo una funzione, ossia una relazione *ovunque definita* e *funzionale*. In particolar modo, concentriamoci sulla proprietà di **funzionalità**:

Proprietà 3.5. *Per ogni insieme S ed ogni coppia di interi N, M deve valere che*

$$\#S = N \ \wedge \ \#S = M \rightarrow N = M.$$

Generalizzando il discorso:

$$\#S = N_1 \wedge \#S = N_2 \wedge \ldots \wedge \#S = N_k \rightarrow N_1 = N_2 \wedge N_1 = N_3 \wedge \ldots \wedge N_1 = N_k$$

Se JSetL trattasse questa proprietà, il vincolo in SF

$$size(S, N) \ \wedge \ size(S, M) \ \wedge \ N \neq M$$

sarebbe ovviamente inconsistente. Purtroppo, anche in questo caso il solver attuale non ci viene incontro: sarà dunque necessaria un'adeguata estensione.

3.3.2 Cardinalità e operazioni insiemistiche

Torniamo per un attimo a parlare di forme risolte. Consideriamo quindi un vincolo *atomico* γ in SF (o equivalentemente in WSF) e supponiamo che in γ compaiano *esclusivamente* insiemi (o variabili logiche instanziate su insiemi). Dalle definizioni 2.15 e 2.16 si può allora osservare come γ possa assumere solamente 4 forme:

1. $X = Y$
2. $X \neq Y$
3. $union(X, Y, Z)$
4. $disj(X, Y)$

Di conseguenza, tutti i rimanenti vincoli atomici di tipo insiemistico (ad esempio *subset* o *inters*...), vengono riscritti in una o più delle precedenti forme.
Osserviamo quindi i legami che possono intercorrere tra tali relazioni insiemistiche e la cardinalità degli insiemi su cui operano.
Nel primo caso, come già osservato nel precedente paragrafo,

$$X = Y \rightarrow \#X = \#Y$$

Nel secondo caso, il vincolo $X \neq Y$ non conduce a considerazioni significative sulla cardinalità di X e Y. Infatti, è possibile avere $\#X = \#Y$ nonostante $X \neq Y$ (ad esempio, $X = \{0\}$ e $Y = \{1\}$).
Analogamente, nemmeno da $disj(X, Y)$ riesco a ricavare informazioni particolarmente rilevanti.
Concentriamoci quindi sul singolo vincolo $union(X, Y, Z)$. Quali osservazioni generali possiamo dedurre circa la cardinalità di X, Y e Z?

Proprietà 3.6. *Siano X, Y, Z insiemi finiti e sia $Z = X \cup Y$, allora valgono le seguenti proprietà:*

(a1) $\#X \leq \#Z$

(a2) $\#Y \leq \#Z$

(a3) $\#Z \leq \#X + \#Y$

Sfortunatamente, nemmeno tali proprietà sono sufficienti per ottenere un vincolo in generale consistente. Infatti, potrei avere anche situazioni in cui $union(X, Y, Z)$ è vincolato alle altre relazioni insiemistiche. Ad esempio, serviamoci del cosiddetto principio di *inclusione-esclusione* limitatamente all'unione di due insiemi (il caso generale, oltre ad avere una forma piuttosto ostica, non servirà ai nostri scopi).

Teorema 3.7 (Principio di inclusione-esclusione)**.** *Siano X ed Y insiemi finiti, allora la cardinalità dell'unione è data da:*

$$\#(X \cup Y) = \#X + \#Y - \#(X \cap Y)$$

Quindi, un vincolo del tipo:

$$union(X, Y, Z) \ \wedge \ inters(X, Y, T) \ \wedge \ size(X, NX) \ \wedge \ size(Y, NY)$$
$$\wedge \ size(Z, NZ) \ \wedge \ size(T, NT) \quad \wedge NZ \ \neq NX + NY - NT$$

sarebbe sicuramente inconsistente, pur essendo in WSF. Tuttavia, definire

una strategia che mi consenta di superare questa limitazione non è per niente facile: la gestione globale diverrebbe assai ardua e inefficiente.
Concentriamoci quindi su altre considerazioni ricavabili dalla cardinalità dell'unione insiemistica:

Proprietà 3.8. *Siano X, Y, Z insiemi finiti e sia $Z = X \cup Y$, allora valgono le seguenti proprietà:*

(b1) $\#X = \#Z \rightarrow X = Z$

(b2) $\#Y = \#Z \rightarrow Y = Z$

(b3) $\#Z = \#X + \#Y \leftrightarrow disj(X, Y)$

Ovviamente, trattare le proprietà (a1), (a2) e (a3) risulterà sicuramente più agevole di quanto lo sia la gestione di (b1), (b2) e (b3) in quanto in esse compaiono anche vincoli insiemistici differenti dall'unione. Nel capitolo 5 vedremo come sarà possibile gestire (*parzialmente*) tali vincoli attraverso l'utilizzo di appositi operatori detti *riscritture*; prima però verrà fatta una piccola digressione sulla gestione degli interi da parte di JSetL.

Capitolo 4

Vincoli elementari sugli interi

Nel capitolo precedente si è notato che per risolvere senza limitazioni troppo restrittive il vincolo *size* è possibile solamente utilizzare il metodo `solve` esteso a tale vincolo, cioè rendere in WSF estesa il vincolo corrente Γ del c.store. Ciò tuttavia comporta la possibilità di avere un vincolo Γ inconsistente anche se non sono presenti vincoli di cardinalità insiemistica (vedi esempi 2.17).
Inolte, il vincolo *size* comporta anche la **gestione di vincoli sugli interi** che il c.solver di JSetL riesce a gestire solo in modo parziale. Per definizione (def. 2.15 e 2.16) infatti:

- Un vincolo in WSF permette qualsiasi operazione tra interi, ma se le variabili in esso non sono instanziate non effettua nessun controllo (si veda ad esempio l'inconsistenza del vincolo c1 dell'esempio 2.17).
- Un vincolo in SF non ammette mai confronti tra interi non inizializzati (casi 6-7 della def. 2.16).

Si noti sin da ora che per avere una soluzione completa e corretta del problema bisognerebbe disporre di un solver che sia in grado di gestire correttamente anche i vincoli sugli interi, in particolare uguaglianze e disuguaglianze linerari [7]. Tuttavia, per poter ottenere un vincolo *size* 'significativo' (ossia, che non sia nè troppo inconsistente e nè troppo limitato) si è cercato di migliorare la situazione con qualche accorgimento che presenteremo nei capitoli successivi.

4.1 Uguaglianze elementari

Definizione 4.1 (Uguaglianza elementare)**.** *Siano* $A_1, A_2 \hat{\in} LV$, $A_3 \hat{\in} LV \hat{\cup} \mathcal{I}$ *tali che* $k(A_i) = 1 \rightarrow VAL(A_i) \hat{\in} \mathcal{I}$ *per* $i = 1, 2, 3$. *Un'**uguaglianza elementare** è un vincolo della forma:*

$$A_1 = A_2 \pm A_3$$

JSetL gestisce correttamente le uguaglianze elementari solamente se (almeno) A_2 e A_3 sono inizializzate: in questo caso posso ottenere immediatamente che $VAL(A_1) = VAL(A_2) \pm VAL(A_3)$ (se $A_3 \hat{\in} \mathcal{I}$, con un piccolo abuso di notazione supporremo che $VAL(A_3) = A_3$).
Tuttavia, basterebbe avere due qualsiasi valori noti $VAL(A_i)$ e $VAL(A_j)$ per ricavare il terzo $VAL(A_k)$ con $i, j, k \in \{1, 2, 3\}$ e i, j, k diversi fra loro, cioè:

$$A_1 = A_2 + A_3 \longleftrightarrow A_2 = A_1 - A_3 \longleftrightarrow A_3 = A_1 - A_2$$

$$A_1 = A_2 - A_3 \longleftrightarrow A_2 = A_1 + A_3 \longleftrightarrow A_3 = A_2 - A_1$$

La definizione di questa semplice regola permetterà ad esempio la risoluzione del vincolo $M = N - 1$ con $\kappa(M) = 1$ e $\kappa(N) = 0$; in questo caso, il vincolo sarà riscritto come $N = M + 1$.
Si noti inoltre che questo accorgimento è stato realizzato 'ad hoc' per i nostri scopi ma potrebbe essere esteso anche ad operazioni quali / e $*$, seppur con uno studio più approfondito (al contrario di + e − infatti queste operazioni non sono *chiuse* al dominio degli interi).
La seguente tabella riassume quindi il comportamento che assumerà JSetL quando si troverà a risolvere una uguaglianza elementare del tipo
$A_1 = A_2 \ \pm \ A_3$ (con lo stesso abuso di notazione utilizzato in precedenza, assumeremo che $\kappa(A_3) = 1$ se $A_3 \hat{\in} \mathcal{I}$).

$\kappa(A_1)$	$\kappa(A_2)$	$\kappa(A_3)$	op	*Risoluzione*
0	0	0	-	Vincolo Irriducibile
0	0	0	+	Vincolo Irriducibile
0	0	1	-	Vincolo Irriducibile
0	0	1	+	Vincolo Irriducibile
0	1	0	-	Vincolo Irriducibile
0	1	0	+	Vincolo Irriducibile
0	1	1	-	$VAL(A_1) = VAL(A_2) - VAL(A_3)$
0	1	1	+	$VAL(A_1) = VAL(A_2) + VAL(A_3)$
1	0	0	-	Vincolo Irriducibile
1	0	0	+	Vincolo Irriducibile
1	0	1	-	$VAL(A_2) = VAL(A_1) + VAL(A_3)$
1	0	1	+	$VAL(A_2) = VAL(A_1) - VAL(A_3)$
1	1	0	-	$VAL(A_3) = VAL(A_1) + VAL(A_2)$
1	1	0	+	$VAL(A_3) = VAL(A_1) - VAL(A_2)$
1	1	1	-	$VAL(A_1) = VAL(A_2) - VAL(A_3)$
1	1	1	+	$VAL(A_1) = VAL(A_2) + VAL(A_3)$

Supponiamo ad esempio di avere
$val(A_1) = 3,\ val(A_2) = X,\ val(X) = \perp,\ val(A_3) = Y,\ val(Y) = 5$
e un vincolo del tipo $A_1 = A_2 + A_3$.
Grazie a tali modifiche riuscirò a ricavare che
$VAL(A_2) = VAL(A_1) - VAL(A_3) = 3 - 5 = -2$.
Perciò, al contrario di prima, se aggiungessi ad esempio il vincolo $A_2 \geq 0$ otterrei giustamente un fallimento.
Se invece avessi $VAL(A_1) = \perp,\ VAL(A_2) = \perp,\ VAL(A_3) = 0$
e un vincolo del tipo $A_1 = A_2 - A_3$ non riuscirei a ricavare che
$VAL(A_1) = VAL(A_2)$ ma rimarrebbe invariata l'uguaglianza $A_1 = A_2 - 0$.
Quindi, se ad esempio aggiungessi il vincolo $A_1 \neq 0$, non otterrei alcun fallimento: il solver di JSetL non giudica (erroneamente) $A_1 = A_2 - 0$ in collisione con $A_1 \neq A_2$.

4.2 Disuguaglianze elementari

Definizione 4.2 (Disuguaglianza elementare)**.** *Siano* $A_1 \hat{\in} LV$ *e* $A_2 \hat{\in} LV \hat{\cup} \mathcal{I}$ *tali che* $k(A_i) = 1 \rightarrow VAL(A_i) \hat{\in} \mathcal{I}$ *per* $i = 1, 2$.
Una ***disuguaglianza elementare*** *è un vincolo della forma:*

$$A_1 \ op \ A_2$$

con $op \in \{<, \leq\}$

In modo analogo alle uguaglianze elementari, JSetL è in grado di dare una risposta corretta solo nel caso in cui $\kappa(A_1) = \kappa(A_2) = 1$.
Si noti che un vincolo della forma $A_1 > A_2$ oppure $A_1 \geq A_2$ può sempre essere trasformato in una disuguaglianza elementare:

- $A_1 > A_2$ diventa $A_2 < A_1$
- $A_1 \geq A_2$ diventa $A_2 \leq A_1$

Ciò consente di ottenere un insieme di vincoli equivalente nel quale però devo trattare solo disuguaglianze di tipo $<$ e $\leq$: a livello implementativo è sicuramente un vantaggio.
Sia quindi γ un vincolo in WSF contenente solamente disuguaglianze elementari; per assicurare la completezza di γ mi basterebbe imporre il rispetto delle proprietà caratteristiche di $<$ e $\leq$ cioè:

Proprietà 4.1. (Proprietà delle relazioni $<$ e $\leq$)

1. *La relazione* $\leq$ *è una relazione d'****ordine*** *e pertanto deve godere delle proprietà:*
   - RIFLESSIVA: $(\forall x)\ x \leq x$
   - ANTISIMMETRICA: $(\forall x, y)((x \leq y \ \wedge y \ \leq x) \rightarrow x = y)$
   - TRANSITIVA: $(\forall x, y, z)((x \leq y \ \wedge y \ \leq z) \rightarrow x \leq z)$
2. *La relazione* $<$ *è una relazione d'****ordine stretto*** *e pertanto deve godere delle proprietà:*
   - ANTIRIFLESSIVA: $(\forall x)\ x \not< x$
   - TRANSITIVA: $(\forall x, y, z)((x < y \ \wedge \ y < z) \rightarrow x < z)$
3. *Le relazioni* $\leq$ *e* $<$ *godono inoltre di proprietà* ***'transitive miste'****, cioè:*

- $(\forall x, y, z)((x < y \ \wedge \ y \leq z\ ) \rightarrow x < z)$
- $(\forall x, y, z)((x \leq y \ \wedge \ y < z\ ) \rightarrow x < z)$

L'inserimento di tali regole nel solver di JSetL sarà trattata nel prossimo capitolo; per ora ci si limiti ad osservare che se tutte le disuguaglianze di γ rispettano queste proprietà allora il vincolo γ è sicuramente consistente.

Tuttavia, nonostante gli accorgimenti adottati permettano un piccolo passo avanti nella gestione dei vincoli sugli interi, esistono ancora parecchi casi non risolti completamente. Ad esempio, i seguenti vincoli:

1. $Z_1 = X + Y \ \wedge \ Z_2 = X + Y \ \wedge \ Z_1 \neq Z_2$
2. $X - Y = 0 \ \wedge \ X \neq Y$
3. $X = X + X \ \wedge \ X > 0$

sono tutti in WSF (e quindi irriducibili) pur essendo chiaramente inconsistenti. Perciò, ribadiamo per l'ennesima volta il concetto: è possibile cercare di migliorare la gestione dei vincoli sugli interi per rendere il vincolo *size* più significativo, tuttavia con il solver attuale non è possibile gestirli in modo completo.

Capitolo 5

Riscritture di vincoli

Nel capitolo 3 sono state introdotte proprietà generali che il vincolo *size* in forma irriducibile dovrebbe soddisfare in relazione ad altri vincoli su interi (3.4 e 3.5) o di tipo insiemistico (3.6 e 3.8), ma che tuttavia non vengono trattate dall'attuale solver.
Nel capitolo 4 sono state riportate le proprietà caratteristiche (4.2) che le disuguaglianze elementari dovrebbero sempre soddisfare; anche in questo caso però il solver attuale non effettua nessun controllo su di esse.
L'obiettivo di questo capitolo è dunque quello di superare tali limitazioni, incrementando di fatto la potenza del solver senza tuttavia trascurare le problematiche di complessità computazionale che potrebbero sorgere. L'approccio utilizzato sarà quindi di tipo *globale*: preso un qualsiasi vincolo γ andranno studiate le possibili relazioni che intercorrono tra tutti i vincoli atomici che lo compongono.
La strategia adottata sarà basata sulle **riscritture**, ossia operazioni che applicate ripetutamente su un certo vincolo γ permettono di ottenere, se possibile, un nuovo vincolo γ' nel quale il solver di JSetL riesca ad individuare (se esistono) inconsistenze precedentemente ignorate .
In generale, le riscritture saranno operazioni **parziali** in quanto non sarà sempre possibile riscrivere con successo il vincolo γ: in questo caso, diremo che la riscrittura termina con **fallimento**.
Si noti che anche i metodi `solve` e `finalSolve` di JSetL implementano in pratica delle riscritture (semplificazioni) sul vincolo corrente del c.store Γ che permettono (se possibile) di ottenere un nuovo vincolo Γ' che sia in WSF o in SF rispettivamente.
Nei paragrafi successivi considereremo una serie di riscritture che, applicate congiuntamente al vincolo corrente del c.store Γ, permetteranno di aumentare la correttezza e la completezza dell'attuale solver.
Prima di prendere esami tali regole, definiamo formalmente il concetto di

riscrittura.

Definizione 5.1 (Riscrittura)**.** *Sia γ un vincolo qualsiasi di JSetL, una* ***riscrittura*** *R su γ è un operazione parziale $R(\gamma)$ tale che, se l'applicazione iterata di R su γ non comporta mai fallimento, R è* ***definitivamente idempotente*** *da un certo indice j in poi. In simboli,*

$$((\forall k \in N) R^{(k)}(\gamma) \neq \texttt{false} \rightarrow)$$

$$(\exists j \in N)(\forall i \in N)(i > j \rightarrow R^{(i)}(\gamma) = R^{(j)}(\gamma))$$

Questa definizione dice che, nell'ipotesi di non avere fallimenti e dopo un certo numero finito j di iterazioni, l'applicazione iterata di una riscrittura su γ mi porta ad ottenere sempre lo stesso vincolo γ'.
Diremo quindi che un generico vincolo γ **soddisfa** la riscrittura R se γ è un **punto fisso** per R, cioè $R(\gamma) = \gamma$.
Nei paragrafi successivi capiremo l'importanza di questa proprietà, ora introduciamo una collezione di riscritture.

5.1 La riscrittura ORD

Nel paragrafo 4.2 si è notato come le disuguaglianze elementari siano soddisfatte se e solo sono soddisfatte le proprietà caratteristiche delle relazioni $<$ e $\leq$. Vediamo ora come è possibile imporre il rispetto di tali proprietà utilizzando l'apposito operatore ORD applicato ad un qualsiasi vincolo γ che contenga generiche relazioni d'ordine o d'ordine stretto (in particolare, potrebbe contenere le relazioni $<$ e $\leq$).

Definizione 5.2 (operatore ORD)**.** *Siano γ un generico vincolo di JSetL e $\preceq$, $\prec$ due generiche relazioni (vincoli) di ordine stretto ed ordine rispettivamente. L'operatore* ***ORD*** *applicato a γ impone ai suoi vincoli atomici il rispetto delle proprietà caratteristiche di $\preceq$ e $\prec$ (illustrate in 4.2 nel caso particolare in cui $\prec$ è la relazione $<$ e $\preceq$ è la relazione $\leq$).*
Il comportamento di $ORD(\gamma)$ è il seguente:

1. Proprietà caratteristiche di $\preceq$

   - *Se trova un vincolo della forma $\boldsymbol{X} \preceq \boldsymbol{X}$ semplicemente lo* ***rimuove*** *da γ (ciò è di fatto un'ottimizzazione, un vincolo di questo tipo è tanto inutile quanto innocuo: la proprietà riflessiva è sempre rispettata).*

- *Se trova un vincolo della forma* $\boldsymbol{X} \preceq \boldsymbol{Y} \wedge \boldsymbol{Y} \preceq \boldsymbol{X}$ *allora lo rimpiazza con* $\boldsymbol{X} = \boldsymbol{Y}$ *per imporre il rispetto della proprietà antisimmetrica.*
- *Se trova un vincolo della forma* $\boldsymbol{X} \preceq \boldsymbol{Y} \wedge \boldsymbol{Y} \preceq \boldsymbol{Z}$ *con* $\boldsymbol{X} \neq \boldsymbol{Z}$ *allora aggiunge a* γ *il vincolo atomico* $\boldsymbol{X} \preceq \boldsymbol{Z}$ *per imporre il rispetto della proprietà transitiva.*

2. Proprietà caratteristiche di $\prec$

- *Se trova un vincolo della forma* $\boldsymbol{X} \prec \boldsymbol{X}$ *allora* γ *è sicuramente* ***inconsistente****:* $ORD(\gamma) =$ `false`
- *Se trova un vincolo della forma* $\boldsymbol{X} \prec \boldsymbol{Y} \wedge \boldsymbol{Y} \prec \boldsymbol{Z}$ *allora aggiunge a* γ *il vincolo atomico* $\boldsymbol{X} \prec \boldsymbol{Z}$ *per imporre il rispetto della proprietà transitiva.*

3. Proprietà 'transitive miste' di $\prec$ e $\preceq$

- *Se trova un vincolo della forma* $\boldsymbol{X} \prec \boldsymbol{Y} \wedge \boldsymbol{Y} \preceq \boldsymbol{Z}$ *allora aggiunge a* γ *il vincolo atomico* $\boldsymbol{X} \prec \boldsymbol{Z}$
- *Se trova un vincolo della forma* $\boldsymbol{X} \preceq \boldsymbol{Y} \wedge \boldsymbol{Y} \prec \boldsymbol{Z}$ *allora aggiunge a* γ *il vincolo atomico* $\boldsymbol{X} \prec \boldsymbol{Z}$

Ovviamente, se in γ non sono presenti disuguaglianze elementari si ha che $ORD(\gamma) = \gamma$. Inoltre, applicando *iterativamente* e *con successo* ORD su γ si arriverà al punto di ottenere sempre lo stesso vincolo γ' in quanto tutte le disuguaglianze elementari di γ' (se esistono) non sono ulteriormente riscrivibili. Ciò è formalizzato nella seguente proposizione:

Proposizione 5.3. *L'operatore ORD è una* ***riscrittura****.*

Dimostrazione. Supponiamo che $(\forall k \in N) ORD^{(k)}(\gamma) \neq$ `false`, cioè non esistono in $ORD^{(k)}(\gamma)$ vincoli del tipo $X < X$, altrimenti la proposizione varrebbe banalmente; va dimostrato che

$$(\exists j \in N)(\forall i \in N)(i > j \rightarrow ORD^{(i)}(\gamma) = ORD^{(j)}(\gamma))$$

Consideriamo un riordinamento dei vincoli atomici di γ (la condizione non è restrittiva per la commutatività della congiunzione logica) raggruppati nel seguente modo:

$$\gamma = \alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge \beta, \text{ dove:}$$

- α_1 è composto da *soli vincoli* della forma $X \leq X$

- α_2 è composto da *soli vincoli* della forma $X \leq Y \wedge Y \leq X$
- α_2 è composto da *soli vincoli* della forma X *op* $Y \wedge Y$ *op* Z con *op* $\in \{<, \leq\}$ e $X \neq Z$
- β è composto da tutti i rimanenti vincoli di γ

L'applicazione $ORD(\gamma)$ diventa quindi $ORD(\alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge \beta)$. Osserviamo allora come opera ORD :

- tutti i vincoli di α_1 vengono rimossi
- tutti i vincoli di α_2 vengono rimossi e vengono aggiunti a β i corrispondenti vincoli della forma $X = Y$
- tutti i vincoli di α_3 rimangono inalterati e vengono aggiungti i corrispondenti vincoli della forma X *op* Z
- tutti i vincoli di β rimangono inalterati e vengono aggiunti i vincoli $X = Y$ derivati dall'applicazione di ORD su α_2

Perciò, $ORD(\gamma) = ORD(\alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge \beta) = ORD(\alpha_3) \wedge \beta\,'$,
con $\beta\,' = \beta \wedge c_1 \wedge c_2 \wedge \ldots \wedge c_m$ dove $c_1, c_2, \ldots, c_m$ sono tanti vincoli di uguaglianza della forma $X = Y$ quanti erano i vincoli di tipo $X \leq Y \wedge Y \leq X$ in α_2.
A questo punto, dimostriamo per induzione su k che $ORD^{(k)}(\gamma) = ORD^{(k)}(\alpha_3) \wedge \beta\,'$.

Passo base: Se k=1, $ORD(\gamma) = ORD(\alpha_3) \wedge \beta\,'$ come appena osservato.

Passo induttivo: Supponendo che $ORD^{(k)}(\gamma) = ORD^{(k)}(\alpha_3) \wedge \beta\,'$ (*ipotesi induttiva*) dimostriamo che $ORD^{(k+1)}(\gamma) = ORD^{(k+1)}(\alpha_3) \wedge \beta\,'$.
Infatti,

$$\begin{aligned} ORD^{(k+1)}(\gamma) &= ORD(ORD^{(k)}(\gamma)) \\ &= ORD(ORD^{(k)}(\alpha_3) \wedge \beta\,') \\ &= ORD(ORD^{(k)}(\alpha_3)) \wedge \beta\,' \\ &= ORD^{(k+1)}(\alpha_3) \wedge \beta\,' \end{aligned}$$

Si noti allora che per dimostrare la proposizione mi basta provare che

$$(\exists j \in N)(\forall i \in N)(i > j \rightarrow ORD^{(i)}(\alpha_3) = ORD^{(j)}(\alpha_3))$$

Il caso peggiore (cioè, quello in cui $ORD(\alpha_3)$ produrre il maggior numero di vincoli) è:

$$\alpha_3 = X_1 \ op \ X_2 \wedge X_2 \ op \ X_3 \wedge \ldots \wedge X_{n-1} \ op \ X_n$$

Infatti, avremo che:

$$ORD(\alpha_3) = \alpha_3 \wedge X_1 \ op \ X_3 \wedge X_2 \ op \ X_4 \wedge \ldots \wedge X_{n-3} \ op \ X_{n-1} \wedge X_{n-2} \ op \ X_n$$

$$ORD^{(2)}(\alpha_3) = ORD(\alpha_3) \wedge X_1 \ op \ X_4 \wedge X_1 \ op \ X_5 \wedge \ldots \wedge X_{n-3} \ op \ X_n \wedge X_{n-4} \ op \ X_n$$

e così via, fino a che dopo $j = n - 2$ iterazioni otterremo:

$$ORD^{(j)}(\alpha_3) = \bigwedge_{0 \leq i < k \leq n} (X_i \ op \ X_k)$$

Con un vincolo di questo tipo, l'applicazione di ORD non ha più effetto in quanto non sarà più possibile dedurre nuove disuguaglianze:

$$(\forall i \in N)(i > j \rightarrow ORD^{(i)}(\alpha_3) = ORD^{(j)}(\alpha_3))$$

□

Consideriamo ad esempio i seguenti vincoli:

1. $\gamma_1 : X \leq Y \wedge Y < 3 \wedge Z \leq Y$
2. $\gamma_2 : X < Y \wedge Y < X$
3. $\gamma_3 : X \leq Y \wedge Y \leq X \wedge X \neq Y$

Vediamo come si comporta la riscrittura ORD:

1. $ORD(\gamma_1) : \gamma_1 \wedge X < 3 \wedge Z < 3$
   $ORD^{(i)}(\gamma_1) = ORD(\gamma_1)$ se $i > 1$
2. $ORD(\gamma_2) : \gamma_2 \wedge X < X$
   $ORD^{(2)}(\gamma_2) =$ `false`;
3. $ORD(\gamma_3) : X = Y \wedge X \neq Y$;
   $ORD^{(i)}(\gamma_3) = ORD(\gamma_3)$ se $i > 1$

Si noti come nell'esempio 1 la riscrittura funziona anche con variabili logiche inizializzate.
L'esempio 2 provoca direttamente fallimento, al contrario dell'esempio 3.
E' importante però osservare come prima della riscrittura γ_3 sia in WSF, mentre successivamente $ORD(\gamma_3)$ non lo è più. Questa è una caratteristica fondamentale della riscrittura: la sua applicazione comporta (se possibile) la generazione di un nuovo vincolo più completo quindi maggiormente significativo.

5.2 La riscrittura $FUNZ$

Come osservato nella proprietà 3.5 della sezione 3.3.1, la relazione di cardinalità deve soddisfare la proprietà di funzionalità. Si era anche notato però come il solver di JSetL non riuscisse a mantenere questa caratteristica, cioè un vincolo del tipo

$$size(S, N) \ \wedge \ size(S, M) \ \wedge \ N \neq M$$

sarebbe in WSF (anzi, addirittura in SF) pur essendo ovviamente inconsistente. Vediamo allora come può una particolare riscrittura permetterci di risolvere questo problema, anche nel caso più generale

$$size(S, N_1) \ \wedge \ size(S, N_2) \ \wedge \ \ldots \ \wedge \ size(S, N_k)$$

Definizione 5.4 (operatore $FUNZ$)**.** *Sia γ un generico vincolo di JSetL. L'operatore* ***FUNZ*** *applicato a γ impone ai suoi vincoli atomici il rispetto della proprietà di funzionalità del vincolo size in* ***forma irriducibile****, sostituendo ogni suo vincolo della forma* ***size(S,N)*** $\wedge$ ***size(S,M)*** *con* ***size(S,N)*** $\wedge$ $\boldsymbol{N = M}$*.*

Proposizione 5.5. *L'operatore $FUNZ$ è una* ***riscrittura****.*

Dimostrazione. Sia $\gamma = \alpha \wedge \beta$ con α contenente solo vincoli della forma $size(S, N) \ \wedge \ size(S, M)$ e β contenente tutti gli altri vincoli (perciò, $FUNZ(\beta) = \beta$). Allora, $FUNZ(\gamma) = FUNZ(\alpha) \wedge \beta\,'$ dove:

- $FUNZ(\alpha) = size(S_1, N_1) \wedge \ldots \wedge size(S_k, N_k)$, con $S_i \neq S_j$ se $i \neq j$ (cioè $FUNZ(\alpha)$ è costituito da soli vincoli *size* su insiemi a due a due *distinti* fra loro)

- $\beta\,' = \beta \wedge c_1 \wedge \ldots \wedge c_n$ dove c_i sono vincoli di uguaglianza derivati dall'applicazione $FUNZ(\alpha)$

Perciò, l'applicazione di $FUNZ$ su $FUNZ(\alpha)$ e β' non ha alcun effetto:

$$\begin{aligned} FUNZ^{(2)}(\gamma) &= FUNZ(FUNZ(\alpha) \wedge \beta') \\ &= FUNZ(\alpha) \wedge \beta' \\ &= FUNZ(\gamma) \end{aligned}$$

Per induzione su i, si può facilmente dimostrare che:

$$(\forall i \in N)(i > 0 \rightarrow FUNZ^{(i)}(\gamma) = FUNZ(\gamma))$$

Quindi $FUNZ$ è una riscrittura (in particolare, $FUNZ$ è anche *idempotente*). □

Consideriamo ad esempio i seguenti vincoli:

1. $\gamma_1 : size(S, N) \wedge size(S, M) \wedge N \neq M$
2. $\gamma_2 : size(S, N) \wedge size(S, M) \wedge size(T, N) \wedge size(T, L) \wedge S \neq T$

Vediamo come si comporta la riscrittura $FUNZ$:

1. $FUNZ(\gamma_1) : size(S, N) \wedge size(S, M) \wedge N = M \wedge N \neq M$
2. $FUNZ(\gamma_2) : size(S, N) \wedge N = M \wedge size(T, N) \wedge N = L \wedge S \neq T$

L'esempio 1 'risolve' il problema presentato a inizio paragrafo: se infatti γ_1 è in WSF, $FUNZ(\gamma_1)$ non lo è più: da $N = M \wedge N \neq M$ anche il solver attuale riesce a ricavare il fallimento.
Nell'esempio 2 si nota come due insiemi distinti S e T possano comunque avere la stessa cardinalità.

5.3 La riscrittura LIM

Occupiamoci ora di un altro problema riguardante l'interazione fra cardinalità e interi, ossia le proprietà di 'limitatezza' 3.4 presentate nel paragrafo 3.3.1.
Come si è potuto notare, queste deduzioni non sono naturalmente implementate nella versione corrente di JSetL: per provvedere a migliorare la situazione sarà quindi necessario l'utilizzo di un apposita riscrittura.

Definizione 5.6 (operatore LIM)**.** *Sia γ un generico vincolo di JSetL. L'operatore* ***LIM*** *applicato a γ impone ai suoi vincoli atomici il rispetto di alcune delle proprietà 3.4 riguardanti la 'limitatezza' del vincolo size in* ***forma irriducibile****.*
Sia quindi $M \hat{\in} \mathcal{LV} \hat{\cup} \mathcal{I}$ tale che $M \hat{\in} \mathcal{LV} \rightarrow VAL(M) \hat{\in} \mathcal{I}$.

$$Se\ k = \begin{cases} M & se\ M \hat{\in} \mathcal{I} \\ VAL(M) & se\ M \hat{\in} \mathcal{LV} \end{cases}$$

allora $LIM(\gamma)$ si comporta nel seguente modo:

- *Se trova un vincolo della forma* ***size(S,N)*** $\wedge$ $\boldsymbol{N \neq 0}$*, aggiunge il vincolo* $\boldsymbol{S \neq \{\}}$
- *Se trova un vincolo della forma* ***size(S,N)*** $\wedge$ $\boldsymbol{N < k}$*, allora:*
  - *Se $k = 1$, aggiunge il vincolo* $\boldsymbol{S = \{\}}$
  - *Se $k \leq 0$, γ è* ***inconsistente****: $LIM(\gamma) =$* `false`
- *Se trova un vincolo della forma* ***size(S,N)*** $\wedge$ $\boldsymbol{N \leq k}$*, allora:*
  - *Se $k = 0$, aggiunge il vincolo* $\boldsymbol{S = \{\}}$
  - *Se $k < 0$, γ è* ***inconsistente****: $LIM(\gamma) =$* `false`
- *Se trova un vincolo della forma* ***size(S,N)*** $\wedge$ $\boldsymbol{k < N}$*, aggiungo il vincolo*
  $S = \{X_1, \ldots, X_k, X_{k+1}|R\}\ \wedge\ \bigwedge_{0 \leq i < j \leq k+1}(X_i \neq X_j)\ \wedge\ size(R, L) \wedge$
  $N = L + k + 1$
- *Se trova un vincolo della forma* ***size(S,N)*** $\wedge$ $\boldsymbol{k \leq N}$*, aggiungo il vincolo*
  $S = \{X_1, \ldots, X_k|R\} \wedge \bigwedge_{0 \leq i < j \leq k}(X_i \neq X_j) \wedge size(R, L) \wedge N = L + k$

Proposizione 5.7. *L'operatore LIM è una* ***riscrittura****.*

Dimostrazione. Come nella dimostrazione della proposizione 5.1, supponiamo che $(\forall k \in N)(RO^{(k)}\gamma) \neq$ `false`.
Sia quindi $LIM(\gamma) = \alpha$, dalla definizione segue che in nessun caso LIM aggiunge dei vincoli della forma

$$size(S, N) \wedge N\ op\ M$$

con $op \in \{\neq, <, \leq\}$, per cui $LIM(\alpha) = \alpha$. Più in generale si dimostra (per induzione su i) che

$$(\forall i \in N)(i > 0 \rightarrow LIM^{(i)}(\gamma) = LIM(\gamma))$$

Quindi LIM è una riscrittura (ed in particolare, come $FUNZ$, è anche *idempotente*). □

Notiamo subito che, come accennato nel paragrafo 3.3.1, non tutte le proprietà 3.4 sono implementate dalla riscrittura LIM.
Infatti, il caso $N \neq k$ lo gestisco solamente se $k = 0$. Questo perchè, a livello pratico, la soluzione
either $N < k$ ***orelse*** $N > k$
proposta in 3.4 comporterebbe un sostanziale aumento della complessità computazionale per valori k (neanche troppo) elevati.
Per lo stesso motivo, nella gestione dei casi $N \leq k$ oppure $N < k$ si è deciso di non utilizzare il costrutto ***either-orelse*** e gestire solo i casi in cui $k = 0$ oppure $k = 1$ rispettivamente. Consideriamo ad esempio i seguenti vincoli:

1. $\gamma_1 : size(S, N) \wedge N < 3$
2. $\gamma_2 : size(S, N) \wedge N \leq \ 2 - 6$
3. $\gamma_3 : size(S, M) \wedge M \leq 2$
4. $\gamma_4 : union(X, Y, Z) \wedge disj(X, Z) \wedge size(X, N) \wedge N \neq 0$

Vediamo come si comporta la riscrittura LIM:

1. $LIM(\gamma_1) : \gamma_1$
2. $LIM(\gamma_2) :$ `false`
3. $LIM(\gamma_3) : \gamma_3 \wedge S = \{X_1, X_2|R\} \wedge X_1 \neq X_2 \wedge size(R, K) \wedge M = K + 2$
4. $LIM(\gamma_4) : \gamma_4 \wedge X \neq \{\}$

Nell'esempio 1, l'operatore LIM non comporta benefici pratici, cioè γ_1 soddisfa già tale riscrittura.
L'esempio 2 causa ovviamente fallimento, essendo $N \leq -4$.
Nell'esempio 3 si ricava che S deve avere almeno 2 elementi e ci si comporta di conseguenza.
Soffermiamoci ora sull'esempio 4: il vincolo $union(X, Y, Z) \wedge disj(X, Z)$ deve portare alla soluzione $X = \{\}$, quindi il vincolo γ_4 è inconsistente (in esso è infatti presente il vincolo $\#X \neq 0$)
Tuttavia, si noti che γ_4 non è solamente in WSF ma anche in SF.

Applicando $LIM(\gamma_4)$, otteniamo un vincolo che è ancora in WSF ma non è più in SF (vìolo la condizione 5 della def. 2.15).
Quindi, i benefici derivati dall'applicazione di una riscrittura non sono limitati solamente all'interazione con la WSF, ma possono riguardare anche la SF.

5.4 La riscrittura UNI

Nel paragrafo 3.3.2 si è visto come il trattamento delle proprietà di cui gode la cardinalità dell'unione insiemistica possono avere varie forme, anche in relazione ad altri vincoli insiemistici come l'uguaglianza o la disgiunzione. Tuttavia, come già accennato, alcune di queste non sono facilmente implementabili. Vediamo allora se e come è possibile superare tali limitazioni.

Definizione 5.8 (operatore UNI)**.** *Sia γ un generico vincolo di JSetL. L'operatore UNI applicato su γ impone ai suoi vincoli atomici il rispetto delle proprietà generali riguardanti la cardinalità del vincolo di unione (cioè i casi* (a1)*,* (a2) *e* (a3) *di 3.6); UNI cerca quindi in γ vincoli atomici di tipo* ***union(X,Y,Z)*** *con $\kappa(X) = \kappa(Y) = \kappa(Z) = 0$ e, se tale vincolo non è ancora stato considerato dalla riscritttura, aggiunge il vincolo:*

$$size(X, NX) \ \wedge \ size(Y, NY) \wedge \ size(Z, NZ)$$
$$\wedge \ NX \leq NZ \ \wedge \ NY \leq NZ \ \wedge \ NZ \leq NX + NY$$

Proposizione 5.9. *L'operatore UNI è una* ***riscrittura****.*

Dimostrazione. Sia $UNI(\gamma) = \alpha$, dalla definizione segue che UNI non aggiunge mai vincoli di unione, quindi:

$$(\forall i \in N)(i > 0 \rightarrow UNI^{(i)}(\gamma) = UNI(\gamma))$$

Perciò UNI è una riscrittura (ed in particolare, come $FUNZ$ e LIM, è anche *idempotente*). □

Consideriamo ad esempio i seguenti vincoli:

1. $\gamma_1 : union(X, Y, X)$ con $\kappa(X) = \kappa(Y) = 0$
2. $\gamma_2 : union(S_1, S_2, S_3) \wedge union(S_3, S_4, S_5)$ con $\kappa(S_i) = 0$ per $i = 1, \ldots, 5$

Vediamo come si comporta la riscrittura UNI:

1. $UNI(\gamma_1) : \gamma_1 \ \wedge \ size(X,N) \ \wedge \ size(Y,M) \ \wedge \ size(X,L) \wedge$
   $N \leq L \ \wedge \ M \leq L \ \wedge \ L \leq N + M$

2. $UNI(\gamma_2) : \gamma_2 \ \bigwedge_{i=1}^{5} size(S_i, N_i) \wedge$
   $N_1 \leq N_3 \ \wedge \ N_2 \leq N_3 \ \wedge \ N_3 \leq N_2 + N_1 \wedge$
   $N_3 \leq N_5 \ \wedge \ N_4 \leq N_5 \ \wedge \ N_5 \leq N_3 + N_4$

Nei due esempi si nota come l'aggiunta di nuovi vincoli causata da UNI apporti una certa pesantezza. Ad esempio, γ_1 genera vincoli inutili, in quanto sicuramente dovrà essere che $L = N$.
Tuttavia se applicassi a $UNI(\gamma_1)$ le riscritture $FUNZ$ e ORD troverei che
$ORD(FUNZ(UNI(\gamma_1))) =$
$\gamma_1 \ \wedge \ size(X,N) \ \wedge \ size(Y,M) \ \wedge \ \ M \leq N \wedge \ N \leq N + M \ \wedge \ N = L.$
Ciò comporta una piccola ottimizzazione di γ_1 (elimino l'inutile vincolo $L \leq M$, anche se purtroppo mantengo l'altrettanto futile $N \leq N + M$).
Se invece a $UNI(\gamma_2)$ applicassi ORD, otterrei che $ORD(UNI(\gamma_2))$ aggiunge nuove disuguaglianze: $N_1 \leq N_5$, $N_2 \leq N_5$ ed $N_3 \leq N_5$.
Possiamo quindi notare che la *composizione* di riscritture può sia ottimizzare un vincolo che renderlo maggiormente completo: questo però può comportare un aumento della sua complessità, soprattutto in termini del numero di vincoli atomici che in esso occorrono.

5.5 La riscrittura INC

Si consideri il vincolo in WSF

$$\gamma = union(X,Y,Z) \ \wedge \ size(X,N) \ \wedge \ size(Z,N)$$

Per la proprietà (b1) di 3.8, dovrebbe essere che $X = Z$ (o equivalentemente $Y \subseteq X$)[1]; tuttavia nessuna riscrittura (o composizione di esse) fin'ora considerata lo impone: di conseguenza, un vincolo $\gamma \wedge X \neq Z$ non provocherebbe alcun fallimento pur essendo inconsistente.
Ovviamente, la stessa cosa vale nel caso *duale*:

$$\gamma = union(X,Y,Z) \ \wedge \ size(Y,N) \ \wedge \ size(Z,N)$$

per la proprietà (b2) di 3.8, dev'essere che $Y = Z$ (o, equivalentemente, $X \subseteq Y$). Serviamoci allora del seguente operatore:

[1]Nella panoramica introduttiva di JSetL si è potuto notare come non tutti i vincoli atomici vengono mantenuti nel c.store: la maggior parte di essi vengono riscritti in funzione di altri vincoli elementari. Tra questi, non fa eccezione il vincolo $subset(Y,X)$ che viene trasformato in $union(Y,X,X)$ o equivalentemente in $union(X,Y,X)$);

Definizione 5.10 (operatore INC)**.** *Sia γ un generico vincolo di JSetL. L'operatore INC applicato a γ impone ai suoi vincoli atomici il rispetto delle proprietà che riguardano la cardinalità del vincolo di unione $union(X, Y, Z)$ con $\kappa(X) = \kappa(Y) = \kappa(Z) = 0$ nel* ***caso particolare*** *in cui $X \subseteq Y$ oppure $Y \subseteq X$. Il comportamento di $INC(\gamma)$ è il seguente:*

- *Se trova un vincolo* ***union(X,Y,Z)*** $\wedge$ ***size(X,N)*** $\wedge$ ***size(Z,N)****, aggiunge il vincolo* ***X=Z****.*
- *Se trova un vincolo* ***union(X,Y,Z)*** $\wedge$ ***size(Y,N)*** $\wedge$ ***size(Z,N)****, aggiunge il vincolo* ***Y=Z****.*

Proposizione 5.11. *L'operatore INC è una* ***riscrittura****.*

Dimostrazione. Sia $INC(\gamma) = \alpha$, dalla definizione segue che INC non aggiunge mai vincoli di unione o di size, quindi:

$$(\forall i \in N)(i > 0 \rightarrow INC^{(i)}(\gamma) = INC(\gamma))$$

Perciò INC è una riscrittura (ed in particolare, come UNI, $FUNZ$ e LIM, è anche *idempotente*). □

Consideriamo ad esempio i seguenti vincoli:

1. $\gamma_1 : union(X, Y, Z) \ \wedge \ size(Y, N) \ \wedge \ size(Z, N) \ \wedge \ Z \neq Y$
2. $\gamma_2 : union(X, Y, Y) \ \wedge \ size(Y, N) \ \wedge \ size(X, N) \ \wedge \ X \neq Y$

Vediamo come si comporta la riscrittura INC:

1. $INC(\gamma_1) : \gamma_1 \ \wedge \ Z = Y$
2. $INC(\gamma_2) : \gamma_2 \ \wedge \ X = Y$

Negli esempi riportati, i vincoli inizialmente sono in WSF pur non essendo corretti; tuttavia l'applicazione di INC fa sì che i vincoli risultanti evidenzino tale inconsistenza.
Si noti inoltre che se anzichè INC avessi utilizzato UNI, non avrei ottenuto niente di significativo: $UNI(\gamma_1)$ e $UNI(\gamma_2)$ sarebbero rimasti in WSF pur essendo logicamente scorretti.

5.6 Il problema della disgiunzione insiemistica

Giunti a questo punto, si può notare come grazie alle riscritture sia stato possibile trattare problemi, precedentementi ingestibili, relativi alle proprietà (a1), (a2) e (a3) di 3.6 e alle proprietà (b1), (b2) di 3.8.
Non ci rimane dunque che affrontare la proprietà (b3), che dice:

$$Se\ Z = X \cup Y,\ allora \quad \#Z = \#X + \#Y \longleftrightarrow disj(X, Y)$$

E' facile osservare che, al contrario delle altre osservate, questà proprietà si basa *esplicitamente* su una *doppia implicazione logica*:

1. $union(X, Y, Z)\ \wedge\ disj(X, Y) \rightarrow \#Z = \#X + \#Y$
2. $union(X, Y, Z)\ \wedge\ \#Z = \#X + \#Y \rightarrow disj(X, Y)$

Si noti come la prima implicazione si potrebbe tranquillamente trattare attraverso un'apposita riscrittura che, qualora trovi un vincolo della forma $union(X, Y, Z)\ \wedge\ disj(X, Y)$, aggiunga $\#Z = \#X + \#Y$.
Il problema sorge dalla seconda implicazione, infatti effettuare un controllo su $union(X, Y, Z)\ \wedge\ \#Z = \#X + \#Y$ non risulterebbe per nulla agevole a causa delle difficoltà di gestione delle uguaglianze elementari in JSetL.
Si potrebbe quindi pensare di implementare la corrispondente proprietà *contronominale*, cioè:

$$ndisj(X, Y)\ \wedge\ union(X, Y, Z) \rightarrow \#Z \neq \#X + \#Y$$

Tuttavia, gestire il vincolo non irriducibile $ndisj(X, Y)$ (corrispondente a $X \cap Y \neq \{\}$) sarebbe ancora più complicato del caso precedente.
Inoltre, si osservi che la proprietà (b3) si basa su un uguaglianza elementare $\#Z = \#X + \#Y$ che può essere gestita correttamente dal solver corrente solo se adeguatamente istanziata.
Una possibile soluzione che risolverebbe questo problema senza far ricorso alle riscritture è la seguente:

either {
$union(X, Y, Z)\ \wedge\ disj(X, Y)\ \wedge\ \#Z = \#X + \#Y$
}
orelse {
$union(X, Y, Z)\ \wedge\ ndisj(X, Y)\ \wedge\ \#Z \neq \#X + \#Y$
}

Tuttavia, l'implementazione di una tale strategia fortemente non-deterministica appare piuttosto improponibile dal punto di vista pratico.
Per tutti questi motivi, si è deciso di **ignorare** la gestione della proprietà (b3), lasciando così il problema aperto ad eventuali sviluppi futuri.

5.7 La riscrittura globale *RG*

Nei paragrafi precedenti si è preso coscienza dei benefici che può portare una riscrittura di un vincolo in termini di completezza e correttezza (anche se ovviamente ciò potrebbe comportare un aumento, comunque limitato, del numero dei suoi vincoli atomici).
In particolare, si è anche accennato come la **composizione** di diverse riscritture possa risolvere situazioni indesiderate.
In questo paragrafo discuteremo quindi una particolare riscrittura, detta *globale*, che usufruirà dell'applicazione congiunta delle precedenti allo scopo di trasformare un vincolo di JSetL nella forma 'più completa possibile' per evitare le inconsistenze viste fino ad'ora.

Definizione 5.12 (operatore *RG*)**.** *Siano ORD, $FUNZ$, LIM, UNI e INC le riscritture presentate nelle definizioni 5.2, 5.4, 5.6, 5.8 e 5.10 rispettivamente. L'operatore di* ***riscrittura globale*** *RG è ottenuto dalla seguente composizione funzionale:*

$$RG = LIM \circ INC \circ ORD \circ FUNZ \circ UNI$$

Proposizione 5.13. *L'operatore RG è una* ***riscrittura****.*

Dimostrazione. Sia γ un generico vincolo, supponiamo che $(\forall k \in N) RG^{(k)}(\gamma) \neq$ `false`; va dimostrato che

$$(\exists j \in N)(\forall i \in N)(i > j \rightarrow RG^{(i)}(\gamma) = RG^{(j)}(\gamma))$$

Lavoriamo quindi iterativamente su γ, ossia ad ogni passo $k = 1, 2, 3, \ldots$ definiamo $\gamma_k = RG^{(k)}(\gamma)$; la tesi sarà dimostrata quando troveremo un certo $q \in N$ tale che:

$$(\forall p \in N)(p > q \rightarrow \gamma_p = \gamma_q)$$

Sia allora

$$\gamma_1 = RG(\gamma) = LIM(INC(ORD(FUNZ(UNI(\gamma)))))$$

Se $\gamma_1 = \gamma$ la dimostrazione termina, altrimenti si ponga

$$\gamma_2 = RG^{(2)}(\gamma) = RG(\gamma_1) = LIM(INC(ORD(FUNZ(UNI(\gamma_1)))))$$

Si noti ora che:

- $UNI(\gamma_1) = \gamma_1$ perchè in generale nessuna riscrittura aggiunge nuovi vincoli *union* significativi per l'applicazione dell'operatore UNI, che quindi non ha effetto su $\gamma_1 = LIM(INC(ORD(FUNZ(UNI(\gamma)))))$.
- $FUNZ(UNI(\gamma_1)) = FUNZ(\gamma_1) = FUNZ(UNI(\gamma))$ perchè l'unica riscrittura che aggiunge vincoli *size* significativi per l'applicazione dell'operatore $FUNZ$ è proprio $UNI(\gamma)$.

Quindi, posto $\alpha = FUNZ(UNI(\gamma))$, si ricava che

$$\begin{aligned} \gamma_2 &= RG(\gamma_1) \\ &= LIM(INC(ORD(FUNZ(UNI(\gamma_1))))) \\ &= LIM(INC(ORD(\alpha)))) \end{aligned}$$

Se $\gamma_2 = \gamma_1$ mi fermo, altrimenti applico nuovamente RG per ottenere

$$\begin{aligned} \gamma_3 &= RG(\gamma_2) \\ &= RG(LIM(INC(ORD(\alpha)))) \\ &= LIM(INC(ORD(LIM(INC(ORD(\alpha))))))) \end{aligned}$$

Questo perchè, a questo punto, l'applicazione di UNI e $FUNZ$ non ha più effetto. Quindi, possiamo notare come nè LIM nè INC aggiungano disuguaglianze elementari, quindi non interferiscono con l'operatore ORD. Osserviamo inoltre come queste due riscritture siano anche *indipendenti* tra loro, cioè non abbiano nessuna interazione. Allora possiamo scrivere:

$$\begin{aligned} \gamma_3 &= LIM(INC(ORD(LIM(INC(ORD(\alpha))))))) \\ &= LIM(INC(LIM(INC(ORD^{(2)}(\alpha))))) \\ &= LIM(LIM(INC(INC(ORD^{(2)}(\alpha))))) \\ &= LIM(INC(ORD^{(2)}(\alpha))) \end{aligned}$$

Generalizzando quindi il discorso per un generico $k > 1$ avremo:

$$\begin{aligned} \gamma_{k+1} &= RG^{(k+1)}(\gamma) = RG(\gamma_k) \\ &= LIM(INC(ORD(\gamma_k))) \\ &= LIM(INC(ORD(LIM(INC(ORD(\gamma_{k-1})))))) \\ \ldots &= LIM(INC(ORD(\ldots(LIM(INC((ORD(\alpha)))\ldots))))) \\ &= LIM(INC(LIM(INC(\ldots(ORD^{(k)}(\alpha))\ldots)))) \\ &= LIM(LIM(\ldots(INC(INC\ldots((ORD^{(k)}(\alpha)))\ldots)\ldots))) \\ &= LIM(INC((ORD^{(k)}(\alpha)))) \end{aligned}$$

Ora, essendo ORD una riscrittura,

$$(\exists j \in N)(\forall j \in N)(i > j \rightarrow ORD^i(\alpha) = ORD^j(\alpha))$$

quindi fissato un tale j avremo che, per ogni naturale i maggiore di j

$$\begin{aligned} \gamma_{i+1} &= RG^{(i+1)}(\gamma) = RG(\gamma_i) \\ \ldots &= LIM(INC(ORD^{(i)}(\alpha))) \\ &= LIM(INC(ORD^{(j)}(\alpha))) \\ \ldots &= RG^{(j+1)}(\gamma) \\ &= \gamma_{j+1} \end{aligned}$$

Perciò, posti $q = j + 1$ (fissato in base al valore di j) e $p = i + 1$ (variabile in base al valore di i, con $i > j$) si ottiene che:

$$p > q \rightarrow i + 1 > j + 1 \rightarrow i > j \rightarrow \gamma_{i+1} = \gamma_{j+1} \rightarrow \gamma_p = \gamma_q$$

□

La riscrittura RG avrà quindi la peculiarità di soddisfare tutte quelle precedentemente definite, rendendo di fatto più completo il procedimento di risoluzione dei vincoli. Ad esempio, se avessi:

$$\gamma = size(S, N) \ \wedge \ N < M \ \wedge \ M \leq 0$$

Otterrei che:

$$\begin{aligned} RG(\gamma) &= LIM(INC(ORD(FUNZ(UNI(\gamma))))) \\ &= LIM(INC(ORD(FUNZ(\gamma)))) \\ &= LIM(INC(ORD(\gamma))) \\ &= LIM(INC(\gamma \ \wedge \ N < 0)) \\ &= LIM(\gamma \ \wedge \ N < 0) \\ &= LIM(\boldsymbol{size(S,N)} \ \wedge \ N < M \ \wedge \ M \leq 0 \ \wedge \ \boldsymbol{N < 0}) \\ &= \texttt{false} \end{aligned}$$

In questo caso noto come solamente ORD e LIM producono vincoli, mentre le altre riscritture non hanno effetto.

Tuttavia, se non avessi usato questa applicazione congiunta utilizzando solamente una tra ORD e LIM non sarei riuscito ad individuare l'inconsistenza di γ.
Ad ogni modo, l'operatore RG acquisisce maggiormente significato se **combinato** con le forme risolte SF e WSF. Infatti, consideriamo:

$$\gamma = union(X, Y, Z) \ \wedge \ size(X, SX) \ \wedge \ size(Z, SZ) \ \wedge \ SZ < SX$$

e applichiamo a γ (inconsistente, in quanto non può essere che $\#X > \#Z$) la riscrittura globale:

$$\begin{aligned}
RG(\gamma) &= LIM(INC(ORD(FUNZ(UNI(\gamma)))))\\
&= LIM(INC(ORD(FUNZ(\\
&= \gamma \ \wedge \ size(X, NX) \ \wedge \ size(Y, NY) \wedge \ size(Z, NZ) \wedge\\
&\quad NX \leq NZ \ \wedge \ NY \leq NZ \ \wedge \ NZ \leq NX + NY))))\\
&= LIM(INC(ORD(\\
&= \gamma \ \wedge \ size(Y, NY) \wedge\\
&\quad NX \leq NZ \ \wedge \ NY \leq NZ \ \wedge \ NZ \leq NX + NY \wedge\\
&\quad NX = SX \ \wedge NZ = SZ)))\\
\ldots &= union(X, Y, Z) \ \wedge \ size(X, SX) \ \wedge \ size(Z, SZ) \ \wedge \ SZ < SX \ \wedge\\
&\quad \wedge \ size(Y, NY) \wedge\\
&\quad NX \leq NZ \ \wedge \ NY \leq NZ \ \wedge \ NZ \leq NX + NY \wedge\\
&\quad NX = SX \ \wedge NZ = SZ
\end{aligned}$$

In questo esempio invece UNI e $FUNZ$ aggiungono nuovi vincoli, mentre ORD, INC e LIM non hanno effetto.
E' immediato notare come l'applicazione RG causi un aumento di vincoli atomici (dai quattro iniziali di γ si è passati ai dodici di $RG(\gamma)$) e non provochi alcun fallimento, seppur il vincolo ottenuto soddisfi RG.
Se però tale vincolo venisse ridotto in WSF, allora grazie alle sostituzioni compiute dal solver di JSetL otterrei il vincolo *logicamente equivalente*:
$\gamma_1 = union(X, Y, Z) \ \wedge \ size(X, SX) \ \wedge \ size(Z, SZ) \ \wedge$
$SZ < SX \ \wedge \ size(Y, NY) \wedge \ SX \leq SZ \ \wedge \ NY \leq SZ \ \wedge \ SZ \leq SX + NY$
A questo punto, applicando due volte RG posso individuare l'inconsistenza del vincolo iniziale $\gamma \equiv \gamma_1$, infatti:

$$\begin{aligned}
RG(\gamma_1) &= LIM(INC(ORD(FUNZ(UNI(\gamma_1)))))\\
\ldots &= LIM(INC(ORD(\gamma_1)))\\
&= LIM(INC(\gamma_1 \ \wedge \ SZ < SZ \ \wedge \ NY < SX))\\
\ldots &= \gamma_1 \ \wedge \ SZ < SZ \ \wedge \ NY < SX
\end{aligned}$$

Essendo $\gamma_1 \neq RG(\gamma_1)$, applico nuovamente RG:

$$\begin{aligned} RG^{(2)}(\gamma_1) &= RG(\gamma_1 \ \wedge \ \boldsymbol{SZ} < \boldsymbol{SZ} \ \wedge \ NY < SX) \\ &= \texttt{false} \end{aligned}$$

Quindi, per individuare **effettivamente** un'inconsistenza, dovrà essere sviluppato un sistema che combini opportunamente le riscritture proposte e le risoluzioni che il solver di JSetL effettuava già in prededenza.
Si noti infatti che la riscrittura RG comporta la produzione di vincoli **corretti** poichè basata su proprietà del dominio degli interi e degli insiemi che si assumono dimostrate.
Tuttavia, se un vincolo γ soddisfa la riscrittura RG non posso affermare che esso sia in generale soddisfacibile. Basti prendere ad esempio

$$\gamma = X < Y \ \wedge \ X = Y$$

In questo caso, $RG(\gamma) = \gamma$ ma ovviamente il vincolo è inconsistente.
Infine, un'importante osservazione riguarda la **commutatività** delle riscritture che compaiono in RG: si potrebbe infatti dimostrare (in modo assai lungo e laborioso...) che componendo in un **qualsiasi ordine** gli operatori UNI, $FUNZ$, ORD, INC, LIM otterrei una riscrittura RG' **equivalente** ad RG. Di conseguenza, l'ordinamento di tali riscritture diventa un dettaglio puramente implementativo.

Capitolo 6

La forma risolta 'parziale'

Nella panoramica introduttiva della libreria JSetL è stato presentato il concetto di vincolo e in particolare ci si è soffermati sulle forme risolte SF e WSF.
Facendo riferimento alle definizioni 2.15 e 2.16, consideriamo quindi un modello insiemistico di tipo *'gerarchico'* che permetta di avere una visione generale riguardo le varie tipologie di vincoli definibili in JSetL. Definiti allora:

- ***V*** l'insieme di tutti i possibili vincoli **definibili** in JSetL.
- ***W*** l'insieme di tutti i vincoli di JSetL **riducibili** in WSF
- ***S*** l'insieme di tutti i vincoli di JSetL **riducibili** in SF

possiamo considerare il seguente grafico:

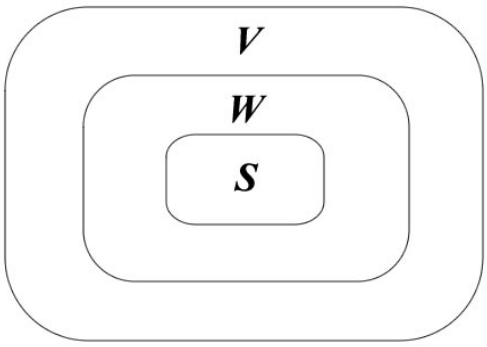


Figura 6.1: Gerarchia dei vincoli di JSetL

Si osservi che questo modello è costituito esclusivamente da **inclusioni strette**: $\boldsymbol{S} \subset \boldsymbol{W} \subset \boldsymbol{V}$. Infatti,

- Un vincolo γ in SF è anche in WSF per definizione: $(\forall \gamma \in \boldsymbol{S}) \gamma \in \boldsymbol{W}$, perciò $\boldsymbol{S} \subseteq \boldsymbol{W}$
- Un vincolo γ in WSF è sicuramente un vincolo di JSetL: $(\forall \gamma \in \boldsymbol{W}) \gamma \in \boldsymbol{V}$, perciò $\boldsymbol{W} \subseteq \boldsymbol{V}$

Da tali considerazioni segue allora che $\boldsymbol{S} \subseteq \boldsymbol{W} \subseteq \boldsymbol{V}$; per completare l'asserto va dimostrato che $\boldsymbol{S} \neq \boldsymbol{W}$ e $\boldsymbol{W} \neq \boldsymbol{V}$. Prendiamo allora i vincoli atomici $\gamma_1 = (X \neq X)$ e $\gamma_2 = (X \leq 0)$ con $X \hat{\in} \mathcal{LV} \wedge \kappa(X) = 0$.
Si noti che:

- $\gamma_1 \in \boldsymbol{V}$ ma $\gamma_1 \notin \boldsymbol{W}$, perciò $\boldsymbol{V} \neq \boldsymbol{W}$.
- $\gamma_2 \in \boldsymbol{W}$ ma $\gamma_2 \notin \boldsymbol{S}$, perciò $\boldsymbol{W} \neq \boldsymbol{S}$.

E' importante osservare che un vincolo γ la cui forma non soddisfi le condizioni di SF (rispettivamente, WSF) può comunque **essere ridotto** in un vincolo γ' in SF (WSF) attraverso il processo di risoluzione dei vincoli del solver attuale di JSetL. Si considerino infatti:

- $\alpha = \ subset(X, Y)$
- $\beta = \ inters(X, Y, Z)$
- $\gamma = \ (X = Y \ \wedge \ X \neq 4)$

Per definizione, nessuno di questi vincoli è in SF (WSF). Tuttavia, ognuno di essi può essere trasformato in SF (WSF) dal solver di JSetL:

- $\alpha' = \ union(X, Y, Y)$
- $\beta' = \ union(S_1, Z, X) \ \wedge \ union(S_2, Z, Y) \ \wedge \ disj(S_1, S_2)$
- $\gamma' = \ Y \neq 4$

Quindi, possiamo affermare che i vincoli $\alpha, \beta, \gamma \in \boldsymbol{S}$ nonostante la loro forma 'originale' non corrisponda alla definizione di SF (WSF).

Effettuiamo ora alcune osservazioni riguardo la soddisfacibilità di un vincolo.
Per la definizione 2.15 di SF possiamo dire che se un vincolo $\gamma \in \boldsymbol{S}$ allora è sicuramente **consistente**; al contrario, se $\gamma \in \boldsymbol{V} \setminus \boldsymbol{W}$ è sicuramente **inconsistente**.

Tuttavia, nel capitolo 3 si è esteso il modello gerarchico dei vincoli assumendo che anche il vincolo $size(S, N)$ in forma irriducibile appartenesse ad $\boldsymbol{S}$ (e quindi anche a $\boldsymbol{W}$ e $\boldsymbol{V}$).
In seguito, si è potuto però notare come questo ampliamento di $\boldsymbol{S}$ potesse condurre alla perdita di soddisfacibilità della SF (si consideri ad esempio il 'solito' vincolo $size(S, N) \ \wedge \ size(S, M) \ \wedge \ N \neq M$).
Inoltre, la WSF aggiungerebbe nuovi casi inconsistenti dovuti alle proprietà che legano $size(S, N)$ a operazioni insiemistiche e tra interi.
Nel capitolo 5 con l'ausilio delle riscritture si è cercato di arginare questo problema, attraverso una serie di trasformazioni che impongano il rispetto di tali proprietà. Ciò nonostante, si è potuto notare come anche la riscrittura globale *RG* necessiti delle semplificazioni in SF e WSF per poter trattare un vincolo in modo più significativo.
Per questo motivo, si è deciso di mantenere le definizioni originali di SF e WSF costruendo una **nuova forma risolta estesa al vincolo *size***, detta *Partial Solved Form*, che permetta di sfruttare sia la riscrittura globale *RG* che le semplificazioni apportate dalle riduzioni in SF e WSF.

6.1 Partial Solved Form

Definizione 6.1 (Partial Solved Form)**.** *Sia $\gamma \in \mathbf{V}$ un generico vincolo di JSetL, allora diremo che γ è in **Partial Solved Form** (PSF) se e solo se sono soddisfatte le seguenti due condizioni:*

(A) *γ soddisfa la **riscrittura globale RG**, cioè $RG(\gamma) = \gamma$*

(B) *ogni vincolo atomico che occorre in γ corrisponde ad una delle seguenti forme:*

- **(B1)** ***1-5** della definizione di **Solved Form**.*
- **(B2)** ***6-7** della definizione di **Weak Solved Form**.*
- **(B3)** ***size(S,N)** con $\kappa(S) = \kappa(N) = 0$.*

La definizione di PSF è sicuramente meno ostica di quanto potrebbe sembrare, infatti:

- La condizione (A) impone solamente il rispetto della riscrittura *RG*
- la condizione (B) di fatto **estende la SF** alle operazioni fra *interi non inizializzati* e ai *vincoli size* in forma irriducibile.

A questo punto, si può notare come le forme (B2) e (B3) comportino la possibile insoddisfacibilità della PSF; tuttavia la condizione (A) rende il vincolo γ in una forma sicuramente più completa rispetto alla WSF.
Allora, detto $\boldsymbol{P}$ l'insieme dei vincoli di JSetL **riducibili** (attraverso applicazioni congiunte e iterate di riscritture e semplificazioni) in *forma parzialmente risolta*, posso definire una nuova gerarchia che tenga conto di $\boldsymbol{P}$, cioè:

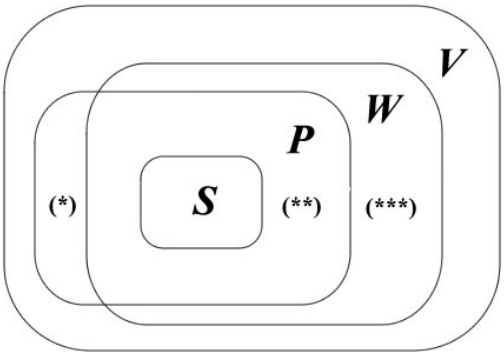


Figura 6.2: Gerarchia estesa dei vincoli di JSetL

Dalla figura si può notare come l'inserimento della PSF nel modello gerarchico determini una forma '*ibrida*' che si pone '*trasversalmente*' tra SF e WSF. Di fatto, $\boldsymbol{P}$ interseca $\boldsymbol{S}$ ed $\boldsymbol{W}$ senza tuttavia essere contenuto in alcuno di questi insiemi. Si osservi infatti che:

(*) L'insieme $\boldsymbol{P} \setminus \boldsymbol{W} \neq \{\}$, ad esempio ***size(S,N)*** con $\kappa(S) = \kappa(N) = 0$ appartiene ad esso.

(**) L'insieme $\boldsymbol{W} \cap \boldsymbol{P} \setminus \boldsymbol{S} \neq \{\}$, ad esempio $\boldsymbol{X} \leq \boldsymbol{0}$ con $\kappa(X) = 0$ appartiene ad esso.

(***) L'insieme $\boldsymbol{W} \setminus \boldsymbol{P} \neq \{\}$, ad esempio $\boldsymbol{X} < \boldsymbol{X}$ con $\kappa(X) = 0$ appartiene ad esso.

6.2 Esempi di risoluzione

Per meglio comprendere il miglioramento apportato dalla PSF, in questa sezione presenteremo una collezione di vincoli **inconsistenti** che il processo di riduzione in forma parziale riesce, al contrario di quanto avveniva precedentemente, a trattare in modo corretto. Ciò significa che, mediante riscritture

iterate e sostituzioni di vincoli, verrà effettivamente rilevata tale inconsistenza.

Esempi 6.2 (Riduzione di un vincolo in PSF)**.**

$$\alpha = size(S, N) \ \wedge \ size(S, M) \ \wedge \ N \neq M \text{ con } \kappa(S) = \kappa(N) = \kappa(M) = 0$$

Si noti innanzitutto che α non soddisfa RG in quanto
$\alpha_1 = RG(\alpha) = \alpha \ \wedge \ N = M$. Al contrario, $RG(\alpha_1) = \alpha_1$, perciò α_1 soddisfa la condizione (A) di 6.1; tuttavia ci si accorge in esso è presente un vincolo del tipo $N = M \ \wedge \ N \neq M$ che in nessun modo è riconducibile ai casi (B1), (B2) o (B3) della PSF: quindi α_1 non soddisfa (B) per cui nemmeno lui è in PSF.
A questo punto, non mi resta che concludere α non è riducibile in PSF: infatti, non posso effettuare ulteriori semplificazioni o riscritture significative da $\alpha_1 = RG(\alpha)$. In altre parole, $\alpha \notin \boldsymbol{P}$.

$$\beta = union(X, Y, X) \ \wedge \ size(X, N) \ \wedge \ size(Y, M) \ \wedge \ N < M$$

L'applicazione iterata di RG su β ci porta ad ottenere un vincolo decisamente ridondante:
$\beta_k = \ldots = RG^{(k)} = \beta \ \wedge \ NX \leq NZ \ \wedge \ NY \leq NZ \ \wedge \ NZ \leq NX + NY \ \wedge$
$N = NX \ \wedge \ M = NY \ \wedge \ N = NZ \ \wedge \ NX = NZ \ \wedge \ X = X$
A questo punto, le sostituzioni applicate dal solver portano a riscrivere più semplicemente β_k come: $\beta \ \wedge \ N \leq M \ \wedge \ M \leq N \ \wedge \ N \leq N + M$
Il vincolo ottenuto soddisfa (B) (è in WSF) ma non altrettanto vale per (A): provo ad applicare nuovamente RG e mi accorgo che $M \leq N \ \wedge \ N < M$ causa la generazione di $N < N$ e quindi un fallimento. Perciò, (A) non è soddisfatta: β non si può ridurre in PSF.

$$\gamma = union(X, Y, Z) \ \wedge \ size(X, N) \ \wedge \ size(Z, N) \ \wedge \ X \neq Z$$

Applicando iterativamente RG, ottengo un vincolo non ulteriormente riscrivibile $\gamma_1 = \ldots = RG^{(k)} = \gamma \ \wedge \ldots \wedge \ X = Z \ \wedge \ X \neq Z$ che soddisfa (A) ma, come visto nel primo esempio, non può soddisfare (B).

Questi sono solo alcuni casi di vincoli inconsistenti che la PSF riesce ad individuare. Altri esempi sono:

- $union(X, Y, Z) \ \wedge \ disj(X, Z) \ \wedge \ size(X, N) \ \wedge \ N \neq 0$
- $size(S, N) \ \wedge \ N < M \ \wedge \ M \leq P \ \wedge \ P < L \wedge \ L \leq 0$
- $inters(X, Y, Z) \ \wedge \ size(X, N) \ \wedge \ size(Z, P) \ \wedge \ P > N$
- $inters(X, Y, Z) \ \wedge \ disj(X, Y) \ \wedge \ size(Z, P) \ \wedge \ P \neq 0$
- $differ(X, Y, Z) \ \wedge \ size(X, N) \ \wedge \ size(Z, P) \ \wedge \ P > N$
- *etc...*

6.3 Problemi aperti

Come dice il nome stesso, la forma risolta *parziale* non risolve tutti i problemi di verifica della soddisfacibilità dei vincoli.
Infatti, pur avendo dimostrato nel precedente paragrafo che la riduzione in PSF comporta un controllo di maggiore consistenza, esistono casi in cui un vincolo è in PSF pur non essendo corretto. Ciò è dovuto a svariati motivi, tra i quali:

- La gestione limitata degli interi, già presentata nel capitolo 4
- Il problema della disgiunzione insiemistica, introdotto nel paragrafo 5.6

Si consideri ad esempio il semplice vincolo $X = X + 1$. Esso, seppur evidentemente inconsistente, è sicuramente in PSF in quanto:

- (A) è rispettata (in particolare, RG non ha alcun effetto)
- (B) è rispettata (in particolare, è rispettata (B2))

Il problema del trattamento dei vincoli sugli interi influenza ovviamente anche la risoluzione del vincolo *size*; sia infatti:

$$S = \{1, 2|R\} \ \wedge \ size(S, N) \ \wedge \ N < 2$$

La risoluzione di questo vincolo (non-deterministica, applico la regola (r4) di 3.2) di *size* porta ad ottenere:
$N < 2 \ \wedge \ 1 \leq N \ \wedge \ size(R, L) \ \wedge \ 1 \notin R \ \wedge \ 2 \notin R$
$\wedge \ L = M - 1 \ \wedge \ M = N - 1 \ \wedge \ 1 \leq M$
Osservando accuratamente il risultato ottenuto, notiamo che in esso è presente il vincolo $N < 2 \wedge \ 1 \leq N$ che dovrebbe (teoricamente) essere ridotto a $N = 1$. A questo punto, $M = N - 1$ implicherebbe $M = 0$, che sarebbe in

ovvia contraddizione con $1 \leq M$.
Perciò, con un solver sugli interi che sia in grado di risolvere equazioni e disequazioni lineari, si potrebbe individuare l'inconsistenza di tale vincolo.

Tuttavia, per ottenere la correttezza di *size* bisognerebbe anche risolvere (possibilmente escludendo la soluzione non-deterministica) il problema della disgiunzione presentato in 5.6. Infatti, vincoli inconsistenti del tipo:

- $union(X, Y, Z) \ \wedge \ disj(X, Y) \ \wedge \ \#Z \neq \#X + \#Y$
- $union(X, Y, Z) \ \wedge \ \#Z = \#X + \#Y \ \wedge \ ndisj(X, Y)$

sono a tutti gli effetti riducibili in PSF e non comportano in alcun modo un fallimento nella risoluzione.

In conclusione, per quanto la PSF possa migliorare in modo significativo la WSF, si può notare come essa sia ancora lontana dalla forma 'corretta' SF.

Capitolo 7

Implementazione

In questo capitolo verranno affrontate le problematiche legate all'implementazione vera e propria del vincolo *size* all'interno della libreria JSetL.
Si osservi fin da ora che ciò verrà comunque trattato in modo piuttosto astratto, cioè senza addentrarci troppo in dettagli di codice che risulterebbero pressochè illegibili per tutti coloro che non conoscessero a fondo l'implementazione di JSetL.
L'obiettivo sarà dunque presentare un modello in *pseudo-codice* che permetta di capire a grandi linee in che modo è stato possibile realizzare le considerazioni ad 'alto livello' introdotte nei precedenti capitoli.

7.1 Il vincolo *size* definito da utente

Facendo riferimento alla sezione 2.4, ricordiamo che JSetL rende possibile la creazione di nuovi vincoli atomici definiti da utente. In questo paragrafo vedremo quindi come sarà possibile implementare il vincolo *size* e la sua risoluzione in accordo con le definizioni 3.1 e 3.2 presentate all'inizio del capitolo 3.
Prima di tutto, è necessario definire a livello utente una nuova classe (che chiameremo `MySizeConstraint`) che estenda la classe astratta di JSetL `NewConstraintClass` e rispetti opportune convenzioni di programmazione.
Illustriamo allora l'interfaccia di `MySizeConstraint`:

```
public class MySizeConstraint extends NewConstraintsClass {

public MySizeConstraint(SolverClass currentSolver);

public Constraint size(LSet S, Integer N);
```

```
  throws ValueNotAllowed

public Constraint size(LSet S, Lvar N);
  throws ValueNotAllowed

public ConstraintsConjunction size(Lvar S,Integer N);
  throws ValueNotAllowed

public ConstraintsConjunction size(Lvar S,Lvar N);
  throws ValueNotAllowed

public void user_code(Constraint c);
  throws Failure, NotDefinedConstraint

private void size(Constraint c);

private void detSize(LSet S, Lvar N, Constraint c);

private void nonDetSize(LSet S, Lvar N, Constraint c);

}
```

Illustriamo ora il significato di tutti i metodi della classe.

```
public MySizeConstraint(SolverClass currentSolver);
```

E' l'unico costruttore di `MySizeConstraint`; di fatto contiene l'unica istruzione `super(s);` che permette di richiamare il costruttore della classe 'padre' `NewConstraintsClass` per realizzare un riferimento al solver corrente `currentSolver` passato come parametro.
Per cui, la creazione di un oggetto di tipo `MySizeConstraint` può avvenire solo nel seguente modo:

```
SolverClass Solver = new SolverClass();
MySizeConstraint myConst = new MySizeConstraint(Solver);
```

I metodi:

- `public Constraint size(LSet S, Integer N)`
  `throws ValueNotAllowed`
- `public Constraint size(LSet S, Lvar N)`
  `throws ValueNotAllowed`

- `public ConstraintsConjunction size(Lvar S,Integer N) throws ValueNotAllowed`

- `public ConstraintsConjunction size(Lvar S,Lvar N) throws ValueNotAllowed`

permettono di costruire vincoli $size(S, N)$ in base al tipo dei parametri S ed N. Si noti che mentre la classe `Constraint` realizza la defizione di *vincolo atomico*, la classe `ConstraintsConjunction` implementa un vincolo vero e proprio, ossia una congiunzione di uno o più vincoli atomici. L'utilizzo di `ConstraintsConjunction` segue dal fatto che, richiamando $size(V, N)$ con $V \hat{\in} \mathcal{LV}$, viene costruito un vincolo equivalente corrispondente alla congiunzione di due vincoli atomici: $size(S, N) \ \wedge \ V = S$.
Osserviamo che ognuno di questi costruttori di vincoli $size(S, N)$ può sollevare l'eccezione `ValueNotAllowed`; questo avviene quando:

1. $S \hat{\in} \mathcal{LV} \wedge \kappa(S) = 1 \wedge VAL(S) \hat{\notin} \mathcal{LS}$
   (violo la condizione **(i)** *della def. 3.1)*

2. $N \hat{\in} \mathcal{LV} \wedge \kappa(N) = 1 \wedge VAL(N) \hat{\notin} \mathcal{I}$
   (violo la condizione **(ii)** *della def. 3.1)*

3. $(N \hat{\in} \mathcal{I} \wedge N < 0) \vee (VAL(N) \hat{\in} \mathcal{I} \wedge VAL(N) < 0)$
   (violo la condizione **(iii)** *della def. 3.1)*

Infatti, nel caso 1 ho che S è un `Lvar` inizializzata ma non instanziata su un `LSet`.
Analogamente, nel caso 2 ho che N è un `Lvar` inizializzata ma non instanziata su un intero.
Nel caso 3 invece N è un intero negativo, che non può in alcun modo corrispondere alla cardinalità di un qualsiasi insieme S.
Quindi, se ora volessi aggiungere un vincolo *size* al c.store del solver corrente, potrei ad esempio utilizzare il seguente codice:

```
LSet S = new ConcreteLSet();
Lvar N = new Lvar();
Solver.add( myConst.size(S,N) );
```

Il comportamento del metodo

```
public void user_code(Constraint c);
  throws Failure, NotDefinedConstraint
```

è già stato illustrato nel paragrafo 2.4 e fa parte di quelle convenzioni da utilizzare per definire un vincolo da utente; in pratica `user_code` associa ad ogni tale vincolo il corrispondente metodo che ne implementa la risoluzione. In questo modo, all'interno della classe `MySizeConstraint` potrei definire anche più di un vincolo, discriminato univocamente in base al suo nome.
Per questo motivo, se `user_code` viene richiamato su di un vincolo `c` non implementato, viene sollevata l'eccezione `NotDefinedConstraint`.
Ad esempio, per quanto visto fin'ora, l'istruzione

```
currentSolver.solve( myConst.size(S,N) );
```

causerebbe tale eccezione in quanto la risoluzione del vincolo `myConst.size(S,N)` non è (ancora) stata definita. Per superare questa limitazione è necessario allora realizzare il seguente metodo:

```
private void size(Constraint c);
```

Al contrario dei metodi considerati in precedenza, si tratta di un metodo `private`, cioè non invocabile all'esterno della classe.
La funzione `size(c)` verrà quindi richiamata dal metodo `user_code(c)` per risolvere (se possibile) il vincolo `c` (che ovviamente sarà della forma $size(S, N)$) passato come parametro.
Osserviamo quindi l'implementazione vera e propria di questo metodo:

```
private void size(Constraint c)  {
  Vector v = c.getMembers();
  LSet S  = (LSet)v.get(0);
  Lvar N = (Lvar)v.get(1);
  if ( S.isGround() || !S.known() )
    detSize(S,N,c);
  else
    nonDetSize(S,N,c);
  return;
}
```

Dapprima, vengono estratti dal vincolo `c` i parametri `S` ed `N`. Quindi, se S è un insieme *ground* oppure *unknown* richiamo il metodo privato `detSize`, che risolve deterministicamente i casi (r1), (r2) ed (r3) della definizione 3.2. In caso contrario, richiamo la funzione `nonDetSize` che invece opera in modo non-deterministico per applicare la regola di risoluzione (r4).
La distinzione di questi due casi anche a livello implementativo è dovuta principalmente a motivi di efficienza (evito di innescare il backtracking).
Vediamo quindi nel dettaglio questi due metodi:

```
private void detSize(LSet S, Lvar N, Constraint c) {
  if ( S.isGround() ) // regola (r1)
    Solver.add( N.eq( S.size() ) ); // N=#S
  else {
    if ( !N.known() ) // regola (r2)
      c.setSolved(false);
    else { // regola (r3)
      Integer k = (Integer)N.getValue(); // S unk., N intera
      Solver.add(S.eq( ConcreteLSet.mkset(k) )); // S={X1,..,Xk}
      if(k>1)
        Solver.add( S.allDifferent() ); // Xi neq Xj
    }
  }
return;
}
```

Se `S` è un insieme *completamente specificato*, allora mi basta aggiungere al solver corrente `Solver` (che è un attributo della classe padre `NewConstraintsClass`) il vincolo atomico `N.eq( S.size() )`. In questo modo impongo che `N=#S` in quanto il metodo di 'utilità' `S.size()` (che non ha nulla a che vedere col vincolo di cardinalità insiemistica) ritorna esattamente il numero di elementi di un insieme *ground*.
Se `S` ed `N` sono unknown, allora il vincolo è in *forma irriducibile*, perciò va lasciato nel c.store di Solver così com'è fino ad una (eventuale) instanziazione di `S` o `N`. Ciò è possibile attraverso l'invocazione del metodo `setSolved(false)` da parte del vincolo `c`.
Se `S` è unknown mentre `N` è inizializzato, opero nel seguente modo:

- dapprima estraggo in `k` l'effettivo valore di `N` col metodo `getValue()` (che, come visto nel capitolo 2, implementa la funzione VAL)
- quindi costruisco un insieme `S={X1,..,Xk}`, costituito da `k` variabili logiche `Xi` non inizializzate, utilizzando il metodo `mkset(k)`.
- infine, se tale insieme ha almeno due elementi, impongo che tutti i suoi elementi siano distinti richiamando l'apposito metodo `allDifferent`.

```
private void nonDetSize(LSet S, Lvar N, Constraint c) {
  Object obj = S.first();
  Lvar e; // estraggo i valori e,R in modo che S = {e|R}
  if (obj instanceof Lvar)
    e = (Lvar)(obj);
  else
```

```
    e = new Lvar(obj);
  LSet R = (LSet)S.sub(); // resto dell'insieme
  Lvar M = new Lvar();
  Solver.add( N.ge(1) ); // N >= 1
  switch( c.getAlternative() ) {
    case 0:
      Solver.addChoicePoint(c);
      Solver.add( e.nin(R) );
      Solver.add( M.eq( N.sub(1) ) ); // M = N - 1
      Solver.add( this.size(R,M) ); // size(R,M)
      break;
    case 1:
      LSet R1 = new ConcreteLSet();
      LSet T = (LSet)( new ConcreteLSet(R1) ).ins(e);
      Solver.add( R.eq(T) ); // R = T = {e|R1}
      Solver.add( e.nin(R1) );
      Solver.add( M.eq( N.sub(1) ) );
      Solver.add( this.size(R1,M) );
      break;
  }
return;
}
```

Il metodo `nonDetSize` implementa il costrutto `either-orelse` corrispondente alla regola (r4) di 3.2, cioè

either {
$S = \{e|R\} \wedge N \geq 1 \wedge e \notin R \wedge M = N - 1 \wedge \ size(R, M)$
}
orelse {
$S = \{e|R\} \wedge N \geq 1 \wedge R = \{e|R1\} \wedge e \notin R1 \wedge M = N - 1 \wedge \ size(R1, M)$
}

Si noti che la funzione ricava la struttura $S = \{e|R\}$ senza passare dall'unificazione insiemistica: vengono utilizzati solamente i metodi di utilità `first` e `sub` della classe `ConcreteLSet`. Quindi, per realizzare il non-determinismo si utilizza lo statement `switch` in modo del tutto analogo a quello presentato nella sezione 2.4 per il metodo `concat`. Infatti, vengono aperte due alternative non-deterministiche (corrispondenti ai due casi di ***either-orelse***) all'interno delle quali vengono aggiunti al solver corrente `Solver` tutti i vincoli necessari.

A questo punto, il vincolo *size* sarebbe pronto per l'uso. Consideriamo infatti il seguente codice:

```
SolverClass Solver = new SolverClass();
MySizeConstraint c = new MySizeConstraint(Solver);
LSet S1 = new ConcreteLSet("S1"); // LSet unknown
Lvar X  = new Lvar("X"); // Lvar unknown
Lvar Y = new Lvar("Y"); // Lvar unknown
Lvar N2 = new Lvar("N2"); // Lvar unknown
LSet S2 = LSet.empty.ins(Y).ins(X); // S2={X,Y}
S2.setName("S2");
LSet R = new ConcreteLSet("R"); // LSet unknown
LSet S3 = ( new ConcreteLSet(R) ).ins('a');
S3.setName("S3"); // S3 = {a|R}
Lvar N4 = new Lvar("N4"); // Lvar unknown
LSet v = LSet.empty.ins(1).ins(2); // v = {2,1}
LSet w = LSet.empty.ins(2).ins(1); // w = {1,2}
LSet S4 = LSet.empty.ins(v).ins(w); // S4 ={ {1,2}, {2,1} }
```

Prendiamo ora in esame *separatamente* le risoluzioni:

1. `Solver.solve( c.size(S1,3) );`
2. `Solver.solve( c.size(S2,N2) );`
3. `Solver.solve( c.size(S3,2) );`
4. `Solver.solve( c.size(S4,N4) );`

Nell'esempio 1, S1 è unknown ma il vincolo `size` impone che S1 = 3, quindi dovrà essere che S1 = $\{X_1, X_2, X_3\}$ con $X_1 \neq X_2, X_2 \neq X_3, X_3 \neq X_1$. Richiamando la `solve`, si otterrà quindi:

```
S1 = {_Lvar_7,_Lvar_6,_Lvar_5}
Store: [ allDifferent({_Lvar_7,_Lvar_6,_Lvar_5}) ]
```

Nell'esempio 2, S2 = {X,Y} con X,Y `Lvar` unknown. Posso allora avere due soluzioni distinte:

1.
```
S2 = {_X,_Y}
N2 = 2
Store: [ _X neq _Y  ]
```

2.
```
S2 = {_X,_X}
N2 = 1
Store: [ ]
```

Nell'esempio 3, S3 = {a|R} con R insieme unknown; il vincolo `size` impone che #S=2, quindi le soluzioni possibili sono:

1.
```
R = {_Lvar_5}
Store: [ a neq _Lvar_5 ]
```

2.
```
R = {a,_Lvar_10}
Store: [ a neq _Lvar_10 ]
```

Nell'esempio 4, S4 = { {1,2}, {2,1} } e N4 è un `Lvar` unknown; la soluzione in questo caso è unica: N4 = 1 in quanto ovviamente {1,2} ≡ {2,1}. Infatti, richiamando la `solve` avremo che:

```
Store: [  ]
N4 = 1
```

7.2 Il vincolo *size* ***built-in***

Nei precedenti esempi si è potuto notare come per la risoluzione del vincolo *size* si sia utilizzato esclusivamente il metodo `solve`, cioè si è presa in considerazione solamente la **WSF estesa al vincolo *size*** definito da utente. Questo perchè, come già osservato nel capitolo 3.1, la riduzione in SF estesa (e quindi l'utilizzo del metodo `finalSolve`) non sarebbe sempre possibile. Ad esempio, la risoluzione (r4) comporterebbe l'utilizzo di vincoli del tipo $N \geq 1, M = N - 1$, ecc... che, nel caso contengano variabili unknown, provocherebbero la perdita della SF (a livello pratico, verrebbe sollevata l'apposita eccezione
`Unitialized_Variable_in_arithmetical_expression`).
Ma questo non è l'unico problema che può sorgere utilizzando il vincolo *size* definito da utente (o *'esterno'*). Infatti, con questo tipo di approccio rimarrebbe completamente irrisolto il problema della **soddisfacibilità globale**: ad esempio, non saprei gestire correttamente risoluzioni del tipo:

```
Solver.solve( c.size(S,N).and( c.size(S,M) ).and( N.neq(M) );
```

oppure

```
Solver.solve( c.size(S,N).and( N.lt(0) );
```

Si noti che per poter trattare questi tipi di problemi bisognerebbe disporre di appositi metodi per la gestione globale dei vincoli del c.store. Al momento, JSetL non fornisce un tale servizio: perciò, si è deciso di adottare un'implementazione **interna** del vincolo *size*, che verrà quindi inserito direttamente nella libreria permettendo così il suo utilizzo come un normale vincolo *built-in* di JSetL. Ovviamente, la realizzazione di tutto ciò presuppone la conoscenza delle classi proprie della libreria quali ad esempio `ConcreteLSet`, `Constraint`, `Enviroment`, `Lvar`, `RewritingConstraintsRules`, `SolverClass` ecc...
Per questo motivo, i dettagli del codice non sono stati riportati: l'unico aspetto rilevante riguarda la variazione di utilizzo del vincolo *size*, in accordo con la nuova definizione. Ad esempio, le invocazioni riportate negli esempi precedenti diventano ora:

1. `Solver.solve( S1.size(3) );`
2. `Solver.solve( S2.size(N2) );`
3. `Solver.solve( S3.size(2) );`
4. `Solver.solve( S4.size(N4) );`

Quindi, non ci sarà più bisogno della dichiarazione

```
MyConstraints c = new MyConstraints(Solver);
```

in quanto con questo tipo di implementazione `size` diventa un metodo pubblico delle classi `LSet` ed `Lvar`.

7.3 I metodi di risoluzione

Facciamo ora un passo indietro alla definizione 2.18 riguardante i metodi `solve()` e `finalSolve()` della classe `SolverClass`. Siano quindi `S` il solver corrente (in termini implementativi, `S` è un istanza di `SolverClass`) e Γ il vincolo contenuto nel c.store di `S`; considerando le notazioni introdotte nel capitolo 6, possiamo affermare che:

- l'invocazione `S.finalSolve()` termina con **successo** se $\Gamma \in \boldsymbol{S}$, altrimenti termina con **fallimento**.

- l'invocazione `S.solve()` termina con **successo** se $\Gamma \in \boldsymbol{W}$, altrimenti termina con **fallimento**.

Analogamente alla definizione del vincolo *size built-in*, l'implementazione di questi due metodi è tutt'altro che intuitiva. Quindi, si è deciso di utilizzare una rappresentazione in pseudo-codice che permetta di comprendere intuitivamente il funzionamento di tali funzioni. Questo ci servirà in seguito quando andremo a definire un particolare metodo che permetta la riduzione in PSF. Ora invece prendiamo in esame il metodo `SolverClass::solve()`

```
public void solve() throws Failure {

    controlloPreliminare();

    controlloSize();

    do {

        semplificaVincoli(Γ);

    } while( Γ ∉ W );

    return;

}
```

Innanzitutto, *controlloPreliminare*() è uno pseudo-metodo che non ha interesse pratico per noi: esso infatti si occupa della gestione di backtracking e variabili quantificicate universalmente.
Il metodo *controlloSize*() invece impedisce l'occorrenza di vincoli di tipo *size* nel c.store; in pratica, si tratta di un semplice controllo lineare sull'insieme dei vincoli di Γ che solleva l'eccezione `Failure` nel caso in esso siano presenti vincoli di questo tipo.
Questo mi serve per impedire l'estensione della WSF ai vincoli *size*, che come ampiamente osservato aggiungerebbe nuove situazioni inconsistenti.
Nel ciclo `do-while` vengono invece effettuate le dovute semplificazioni su tutti i vincoli atomici di Γ, fintanto che Γ non risulti ridotto in WSF. Se non fosse possibile effettuare tali trasformazioni di Γ, verrebbe sollevata l'eccezione `Failure`; in caso contrario al termine della chiamata di `solve()` otterremo che $\Gamma \in \boldsymbol{W}$.

Vediamo quindi come funziona l'altro metodo di risoluzione
`SolverClass::finalSolve()`

```
public void finalSolve() throws Failure {

    controlloInteri();

    solve();

    controlloUnione();

    return;

}
```

Inizialmente viene effettuato il test *controlloInteri*() che, con un funzionamento analogo a *controlloSize*, permette di evitare in Γ l'occorrenza di confronti fra interi non inizializzati (condizioni 6-7 della WSF): se tali vincoli fossero presenti nel c.store verrebbe sollevata l'eccezione
`Uninitialized_Variable_in_arithmetical_expression`.
Quindi, il vincolo corrente Γ viene ridotto in WSF con la chiamata di `solve()` (ovviamente, se $\Gamma \notin \boldsymbol{W}$ verrà sollevata l'eccezione `Failure`).
A questo punto, facendo riferimento alle definizioni 2.15 e 2.16, si può notare come le prime quattro condizioni di SF e WSF siano identiche. La condizione 5, relativa al vincolo $union(Z_1, Z_2, Z_3)$ con Z_i unknown, è invece leggermente differente:

- Nella SF impongo che $Z_1 \not\equiv Z_2$ e che non esistano vincoli della forma $Z_i \neq E$, con $E \hat{\in} \mathcal{L}$
- Nella WSF tale regola è rilassata: si impone solamente che $Z_1 \not\equiv Z_2$

Perciò, la chiamata di *controlloUnione*() permetterà di verificare che nel vincolo Γ (che a questo punto è in WSF) non siano presenti vincoli della forma $Z_i \neq E$. Si tenga tuttavia presente che questa funzione opera in modo *non-deterministico* e potrebbe quindi causare inefficienza. Infatti, se aggiungessi al c.store il vincolo:

```
Solver.add( X.subset(Y).and( Y.subset(X) ).and( X.neq(Y) ) );
```

il solver di JSetL lo ridurrebbe in $union(X, Y, Y) \wedge union(Y, X, X) \wedge X \neq Y$.
A questo punto, si può notare come un tale vincolo sia in WSF: la chiamata di `solve()` avrebbe successo.

Al contrario, la presenza di $X \neq Y$ comporterebbe la violazione della condizione 5 della SF: la chiamata di `finalSolve()` provocherebbe quindi l'eccezione `Failure`.
Tuttavia, essendo il test *controlloUnione*() non-deterministico, l'individuazione del fallimento non è immediata. Per questo motivo, il controllo della condizione 5 è posticipato alla fine del metodo: se possibile, evito di attivare inutili procedure non-deterministiche.

7.3.1 Il metodo `partialSolve`

Dopo aver dato le definizioni dei metodi `solve` e `finalSolve` per la riduzione del vincolo Γ in WSF e SF, non ci resta che definire una corrispondente funzione che permetta (se possibile) la trasformazione di Γ in PSF. Dalla definizione 6.1 possiamo notare che le uniche variazioni apportate dalla PSF rispetto alle SF e WSF riguardano la gestione del vincolo *size* e della *riscrittura globale*RG.
Nel capitolo 5 la riscrittura globale RG è stata definita a livello piuttosto astratto, senza preoccuparci dei dettagli implementativi. In questo paragrafo supporremo quindi di avere a disposizione una pseudo-funzione del tipo *riscritturaGlobale*(γ) che, preso come parametro un qualsiasi vincolo $\gamma \in \boldsymbol{V}$, lo modifica sostituendolo con l'applicazione $RG(\gamma)$
(cioè, $\gamma \leftarrow RG(\gamma) = LIM(INC(ORD(FUNZ(UNI(\gamma))))))$).
L'implementazione di LIM, INC, ORD, $FUNZ$ e UNI è affidata a corrispondenti metodi che non descriveremo in questa sede; limitiamoci solamente a ricordare che l'*ordine* con cui vengono applicati tali operatori è ininfluente.
Per permettere che in Γ possa occorrere anche un vincolo *size*, basterà semplicemente ignorare lo pseudo-metodo *controlloSize* utilizzato nei metodi `solve` e `finalSolve`.
A questo punto, nella classe `SolverClass` posso definire il metodo pubblico `partialSolve()`, la cui invocazione termina con successo se Γ può essere ridotto in PSF (cioè, $\Gamma \in \boldsymbol{P}$) altrimenti termina con fallimento (cioè, $\Gamma \notin \boldsymbol{P}$).
Vediamo quindi lo pseudo-codice di tale funzione:

```
public void partialSolve() throws Failure {

    controlloPreliminare();

    do {
```

$semplificaVincoli(\Gamma)$;

$riscritturaGlobale(\Gamma)$;

} `while(` $\Gamma \in \boldsymbol{W}$ $''esteso''$ $\wedge$ $RG(\Gamma) = \Gamma$);

$controlloUnione()$;

`return;`

}

Il metodo $controlloPreliminare()$ è il medesimo utilizzato in `solve`. Nel ciclo `do-while` si può notare come, oltre al metodo $semplificaVincoli(\Gamma)$ che il solver utilizza per operare le sostituzioni dei vincoli di Γ, sia presente anche il metodo $riscritturaGlobale(\Gamma)$. In questo modo, all'interno del ciclo vengono operate congiuntamente semplificazioni e riscritture globali fino ad ottenere, se possibile (altrimenti sollevo la 'solita' eccezione `Failure`), un vincolo Γ che:

- sia in WSF estesa ai vincoli *size*
- rispetti la condizione (A) di 6.1, cioè $RG(\Gamma) = \Gamma$

Ora, osservando attentamente le definizioni, si può notare come un tale vincolo non è assicurato essere in PSF perchè non è garantito il rispetto della condizione (B) di 6.1. Infatti, potrei violare la condizione 5 della SF: per questo motivo, come nella `finalSolve`, al termine del metodo viene aggiunta la funzione $controlloUnione()$.

Per concludere, vediamo qualche esempio di utilizzo pratico del metodo `partialSolve`.

```
SolverClass Solver = new SolverClass();
Lvar N = new Lvar("N"); \\ Lvar unknown
Lvar M = new Lvar("M"); \\ Lvar unknown
Lvar L = new Lvar("L"); \\ Lvar unknown
Lvar K = new Lvar("K"); \\ Lvar unknown
Lvar N1 = new Lvar("N1"); \\ Lvar unknown
Lvar N2 = new Lvar("N2"); \\ Lvar unknown
Lvar N3 = new Lvar("N3"); \\ Lvar unknown
```

```
LSet S = new ConcreteLSet("S"); \\ LSet unknown
LSet X = new ConcreteLSet("X"); \\ LSet unknown
LSet Y = new ConcreteLSet("Y"); \\ LSet unknown
LSet Z = new ConcreteLSet("Z"); \\ LSet unknown
LSet A = new ConcreteLSet("A"); \\ LSet unknown
LSet B = new ConcreteLSet("B"); \\ LSet unknown
LSet C = new ConcreteLSet("C"); \\ LSet unknown
LSet D = new ConcreteLSet("D"); \\ LSet unknown
```

Consideriamo quindi *separatamente* le seguenti istruzioni:

1. `Solver.add( S.size(N1).and( S.size(N2) ).and( S.size(N3) ));`
2. `Solver.add( inters(X,Y,Z).and( disj(X,Y) ).`
   `and( Z.size(N) ).and( N.neq(0) );`
3. `Solver.add( A.subset(B).and( A.size(L) ).and( B.size(M) ).`
   `and( L.ge(K) ).and( M.le(K) );`
4. `Solver.add( C.subset(D).and( D.subset(C) ).and( C.neq(D) ));`

e invochiamo, dopo ognuna di esse, il metodo `Solver.partialSolve();`

Nell'esempio 1, la riscrittura RG impone che $N1 = N2 = N3$, quindi le sostituzioni applicate dal solver riducono il c.store a

```
Store: [ size(_S,_N1) ]

N1 = unknown

N2 = _N1

N3 = _N1
```

Nell'esempio 2, si può notare l'inconsistenza del vincolo

$$Z = X \cap Y \ \wedge \ disj(X,Y) \ \wedge \ \#Z \neq 0$$

Tuttavia, se non venisse esplicitamente aggiunto il vincolo $Z \neq \{\}$, non si arriverebbe al fallimento. Attraverso l'ausilio della riscrittura RG (ed in particolar modo a LIM) tale vincolo viene aggiunto e, grazie al metodo *controlloUnione*(), viene sollevata l'eccezione `Failure`.

Nell'esempio 3, il vincolo $A \subseteq B \wedge \#A = L \wedge \#B = M \wedge L \leq K \wedge M \leq K$

deve condurre alla soluzione $A = B$ (essendo $A \subseteq B$ e quindi $\#A \leq \#B$).
Ciò è implementato correttamente da `partialSolve` che, dopo le dovute trasformazioni, ottiene:

```
Store: [ size(_A,_L),  ]

_K = _L

_M = _L

X = unknown

Y = _X
```

Si noti tuttavia che ciò funziona efficientemente poichè `K` è una `Lvar` unknown: se invece fosse istanziata su un intero k, allora ciò porterebbe alla **generazione** di due insiemi del tipo $A = \{a_1, \ldots, a_k\}$ e $B = \{b_1, \ldots, b_k\}$ il cui processo di unificazione (necessario per soddisfare l'uguaglianza insiemistica $A = B$) causerebbe inefficienza anche per valori di k molto piccoli.

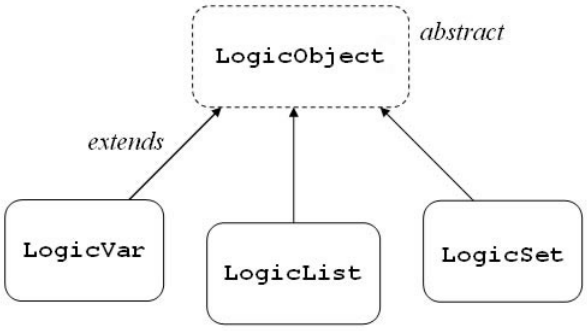
Nel punto 4, viene ripreso il problema, già presentato nel precedente paragrafo, del vincolo $C \subseteq D \ \wedge \ D \subseteq C \ \wedge \ C \neq D$. Come si era potuto notare, un tale vincolo è riducibile in WSF ma non in SF.
Tuttavia, per individuare l'inconsistenza il metodo `finalSolve` deve ricorrere al non-determinismo.
Al contrario, `partialSolve` riesce a risolvere tale vincolo semplicemente sfruttando le riscritture.
Infatti, l'operatore RG riuscirà dopo un certo numero di iterazioni a ricavare che $\#C = \#D$ e quindi $C = D$ (essendo $C \subseteq D$). Ovviamente, la presenza nel c.store di $C = D$ e $C \neq D$ provocherà l'eccezione `Failure` senza bisogno di attivare nessun meccanismo non-deterministico.

Capitolo 8

Conclusioni e lavori futuri

Il lavoro di tesi è cominciato con lo studio di JSetL, partendo dalle conoscenze preliminari acquisite nei corsi di linguaggi dichiarativi e object-oriented.
Si sono quindi definiti il vincolo di cardinalità insiemistica e le proprietà caratteritiche che esso deve soddisfare.
Tuttavia, si è notato come garantire tali proprietà non sia per nulla banale: la presenza del dominio degli interi implica una serie di complicazioni che non sono state del tutto risolte.
Ciò nonostante, le tecniche utilizzate per limitare questi problemi possono risultare un buon punto di partenza per eventuali ampliamenti futuri.
Indichiamo quindi alcune proposte di sviluppo del lavoro svolto.

Per quanto riguarda il package JSetL, risulterebbe utile un processo formalizzato di *ingegnerizzazione*, mirato a incrementarne qualità, efficienza, ri-usabilità e comprensibilità. Una tale ristrutturazione potrebbe a grandi linee seguire una metodologia di tipo '*Unified Process*' organizzata in fasi successive (quali ad esempio analisi, progettazione, implementazione, test e documentazione) che permetta di ottenere un consistenete miglioramento del software in questione.
Ad esempio, la suddivisione degli *oggetti logici* in variabili, liste ed insiemi logici presentata all'inizio del capitolo 2 è puramente teorica. All'interno della libreria infatti `Lvar`, `Lst` ed `LSet` costituiscono di fatto tre classi indipendenti che, seppur abbiano parecchie caratteristiche in comune, non sono in alcun modo in relazione fra loro. In riferimento ai princìpi della programmazione object-oriented ed in particolar modo alla relazione di ereditarietà fra classi, si potrebbe quindi definire uno schema gerarchico del tipo



In questo modo anche dal punto di vista pratico impongo che variabili, liste e insiemi logici (indicati nello schema con `LogicVar`, `LogicList` e `LogicSet` rispettivamente) *siano effettivamente* degli oggetti logici attraverso la specializzazione della classe astratta `LogicObject`.
Un altro miglioramento della libreria potrebbe riguardare la *definizione di vincoli da utente*. Infatti, come si è potuto notare per *size*, l'introduzione di un nuovo vincolo (e la conseguente estensione ad esso di SF e WSF) potrebbe causare l'insoddisfacibilità globale del c.store. Per risolvere questo problema si potrebbe offrire all'utente la possibilità di manipolarne i vincoli atomici (anche se la cosa non mi sembra particolarmente corretta) oppure riservare ai vincoli definiti da utente una forma risolta apposita del tipo *User Solved Form* che non vada ad intaccare SF e WSF.

Per quel che concerne il vincolo *size*, gli sviluppi possono risultare molteplici. Innanzitutto si potrebbe cercare di migliorare ulteriormente il trattamento degli interi non inizializzati, facendo magari riferimento ad altri solver che gestiscono correttamente questi vincoli.
Quindi, si potrebbe ottimizzare e completare il sistema di riscritture introdotto per incrementare la consistenza di *size*.
L'obiettivo ideale da raggiungere (anche a livello puramente teorico) sarebbe infatti ottenere la *piena soddisfacibilità* della PSF, ossia la risoluzione di tutti i problemi aperti presentati in questa tesi.
Tuttavia ciò appare tutt'altro che banale, in quanto la dimostrazione di una tale proprietà coinvolge due domini CLP *distinti*: gli interi e gli insiemi.

Bibliografia

[1] A. Dovier, C. Piazza, E. Pontelli, G. Rossi
Sets and constraint logic programming
ACM TOPLAS 2000; 22(5):861-931.

[2] A. Dal Palù, A. Dovier, E. Pontelli, G. Rossi
Integrating finite domain constraints and CLP with sets.
ACM Press: New York, 2003; 219-229.

[3] Francisco Azevedo
Cardinal: A Finite Sets Constraint Solver
Constraints 2007; 12:93-129.

[4] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo.
JSetL: a Java library for supporting declarative programming in Java.
Software Practice & Experience 2007; 37:115-149.

[5] JSetL Home Page.
`http://prmat.math.unipr.it/%7egianfr/JSetL/index.html`

[6] Delia Di Giorgio.
Gestione di insiemi ed operazioni insiemistiche in Java tramite l'integrazione tra la libreria JSetL e l'interfaccia Set di Java.
`http://www.cs.unipr.it/Informatica/Tesi/Delia_DiGiorgio_20070426.pdf`

[7] Krzysztof R. Apt.
Principles of Constraint Programming.
Cambridge University Press, 2003.

[8] Agostino Dovier, Roberto Giacobazzi.
Fondamenti dell'Informatica: Linguaggi formali, Calcolabilità e Complessità.
`http://users.dimi.uniud.it/ãgostino.dovier/DID/dispensa.pdf`

[9] Java Platform, Standard Edition 6: API Specification.
`http://java.sun.com/javase/6/docs/api/`