UNIVERSITÀ
DI PARMA

Dipartimento di Scienze
Matematiche, Fisiche e Informatiche

Corso di Laurea in Informatica

Trattamento di vincoli su insiemi e relazioni binarie nella libreria Java JSetL

Dealing with constraints on sets and binary relations in the Java library JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Michael Cobianchi

Anno Accademico 2017/2018

Deicato ai miei genitori
per avremi sostenuto
e permesso di raggiungere
questo importante traguardo.

Indice

1 Introduzione

Le relazioni binarie sono un concetto matematico chiave per la specifica e la verifica del software. Avere strumenti in grado di trattare in modo automatico operazioni su relazioni binarie potrebbe essere di interesse per vari settori, dalla dimostrazione di teoremi alla programmazione con vincoli.
Le funzioni parziali sono un sottoinsieme delle relazioni binarie, che a loro volta sono insiemi di coppie ordinate; pertanto, sembra naturale provare a codificare le relazioni binarie in termini di insiemi. In [Cristià Rossi 2013], è stato mostrato come le funzione parziali possano essere codificate in termini di insiemi su CLP($\mathcal{SET}$) un linguaggio CLP (Constraint Logic Programming) dotato di una procedura decisionale in grado di risolvere le formule del primo ordine senza quantificatori su un universo di insiemi finiti. Nonostante questa codifica sembri abbastanza semplice, ci si accorge immediatamente come operatori di base su relazioni binarie, come dominio, rango (codominio) e composizione, non possano essere espressi da formule di CLP($\mathcal{SET}$).
In [Cristià Rossi 2016] viene mostrato come estendere CLP($\mathcal{SET}$) in modo tale che relazioni e funzioni parziali siano entità primitive del linguaggio. Il nuovo linguaggio logico, chiamato $\mathcal{L}_{\mathcal{BR}}$, offre insiemi, relazioni binarie e funzioni parziali, con le relative operazioni di base su essi. $\mathcal{L}_{\mathcal{BR}}$ consente di combinare liberamente e naturalmente insiemi e relazioni binarie; in particolare, le relazioni binarie possono essere manipolate dai classici operatori definiti su insiemi (ad esempio unione, intersezione, ecc.), oltre che dai tipici operatori relazionali (ad es. dominio e rango di una relazione, composizione, relazione inversa, ecc...).
JSetL è una libreria Java che combina il paradigma di programmazione object-oriented di Java con i concetti dei linguaggi CLP, come variabili logiche, liste, unificazione, risoluzione dei vincoli e non determinismo. In particolare la libreria fornisce l'implementazione delle variabili logiche tramite la classe `LVar` e degli insiemi logici tramite la classe `LSet`. La libreria, inoltre, fornisce le apposite classi per l'implementazione di vincoli su variabili e insiemi logici, oltre che del risolutore per tali vincoli. In definitiva, JSetL fornisce una implementazione pressoché

completa del linguaggio CLP($\mathcal{SET}$) all'interno di Java.
In questo lavoro di tesi verrà estesa la libreria JSetL per implementare il linguaggio $\mathcal{L}_{\mathcal{BR}}$ al suo interno. Sfruttando il meccanismo dell'ereditarietà tra classi offerto da Java, viene creata la classe `LRel`, per l'implementazione delle relazioni binarie, come estensione della classe `LSet`. Sfruttando il medesimo meccanismo, viene creata la classe `LMap`, per l'implementazione delle funzioni parziali, come estensione della classe `LRel`, rispettando così la definizione stessa di relazione binaria e funzione parziale. La libreria JSetL viene inoltre fornita della classe `LPair`, per la realizzazione delle coppie logiche di cui relazioni binarie e funzioni parziali sono composte, e della classe `RwRulesBR` contenente le implementazioni delle regole di riscrittura dei vincoli specifici per relazioni binarie e funzioni parziali.

Il resto di questo elaborato di tesi è così strutturato:

- **Capitolo 2**: In questo capitolo verranno presentate la sintassi e la semantica del linguaggio CLP($\mathcal{SET}$) con la procedura di risoluzione adottata dal solver $SAT_{\mathcal{SET}}$; verrà prensentato inoltre il linguaggio $\mathcal{L}_{\mathcal{BR}}$ su relazioni binarie e funzioni parziali con la sintassi utilizzata per la loro rappresentazione e la semantica e sintassi dei nuovi vincoli aggiunti su di esse.

- **Capitolo 3**: In questo capitolo verranno dapprima richiamate le classi che realizzano l'implementazione nella libreria JSetL del linguaggio CLP($\mathcal{SET}$) con il relativo risolutore di vincoli; successivamente verranno presentate e descritte in dettaglio le nuove classi `LPair`, `LRel` ed `LMap` per l'implementazione del linguaggio $\mathcal{L}_{\mathcal{BR}}$ nella libreria JSetL.

- **Capitolo 4**: In questo capitolo verranno presentate le regole di riscrittura per i nuovi vincoli su relazioni binarie (`LRel`), mostrando per ciascuna di esse la relativa implementazione con qualche esempio di utilizzo dei vincoli su `LRel`.

- **Capitolo 5**: In questo capitolo verranno presentate le regole di riscrittura e la loro implementazione con qualche esempio di utilizzo dei vincoli su funzioni parziali (`LMap`) più alcuni dettagli implementativi relativi al lavoro svolto.
- **Capitolo 6**: In quest'ultimo capitolo si trarranno alcune considerazioni sul lavoro svolto e si accennerà ad eventuali lavori ed argomenti futuri da affrontare.

2 Un linguaggio a vincoli su insiemi e relazioni binarie

In questo capitolo vengono presentati i linguaggi CLP($\mathcal{SET}$) e $\mathcal{L}_{\mathcal{BR}}$.
Il linguaggio a vincoli CLP($\mathcal{SET}$), già implemetato nella libreria Java JSetL, è definito tramite la sua sintassi, la semantica (informale) ed i vincoli su insiemi logici [5].
Il linguaggio a vincoli $\mathcal{L}_{\mathcal{BR}}$ include ed estende il linguaggio a vincoli CLP($\mathcal{SET}$) ed è dato definendo i vincoli su relazioni binarie, in accordo con la definizione data in [4].

2.1 Il linguaggio CLP($\mathcal{SET}$)

2.1.1 Sintassi

La sintassi del linguaggio CLP($\mathcal{SET}$) si basa sui seguenti insiemi di simboli:

- $\mathcal{F}$ è un insieme di costanti e simboli di funzione così definito:
  - $\emptyset \in \mathcal{F}$
  - $\{\cdot \mid \cdot\} \in \mathcal{F}$ e $int \in \mathcal{F}$, simboli relazione binaria
  - $Z \subset \mathcal{F}$ e $F_Z \subset \mathcal{F}$, dove Z l'insieme delle costanti numeriche intere e F_Z l'insieme degli operatori aritmetici di base (+, -, $*$, /).
- $\prod_C = \{=, in, un, disj, \leq, size, set, integer\}$, insieme di simboli di predicato
- $\mathcal{V}$ è un insieme numerabile di simboli di variabile

I $\mathcal{SET}$-*termini* sono termini costruiti a partire dai simboli di $\mathcal{F}$ e $\mathcal{V}$ nel modo usuale. Ad esempio, i seguenti, sono termini insiemistici ($set-terms$):

- $\{1 \mid \emptyset\} \equiv \{1\}$
- $\{1 \mid \{2 \mid \emptyset\}\} \equiv \{1, 2\}$
- $\{1 \mid \{2 \mid X\}\} \equiv \{1, 2 \mid X\}$
- *etc...*

I $\mathcal{SET}$-*costraints* atomici (o primitivi) sono predicati i cui simboli sono definiti in $\prod_C$, come ,per esempio:

- $X = 1$
- $un(\{1\}, \emptyset, R)$

Le $\mathcal{SET}$-$formula$ (o $\mathcal{SET}$-*costraints* composti) sono combinazioni ($\wedge(and)\backslash\vee(or)$) di $\mathcal{SET}$-*constraint* primitivi. Le seguenti formule sono esempi di $\mathcal{SET}$-*constraint* composti:

- $1 \ in \ R \ \wedge \ 1 \ nin \ S \ \wedge \ inters(R, S, T) \ \wedge \ T = \{X\}$
- $inters(R, S, T) \ \wedge \ size(T, N) \ \wedge \ N \leq 2$

2.1.2 Semantica informale

La semantica intuitiva dei vari simboli in $\sum_{\mathcal{SET}}$ è la seguente:

- $\emptyset$ rappresenta l'insieme vuoto
- $\{\cdot \mid \cdot\}$ rappresenta il costruttore di insiemi così definito:
  $\{t \mid s\} = \{t\} \cup$ s
- *int* è il costruttore di intervalli definito come segue:

$$int(m, n) = \begin{cases} \emptyset & \text{se } m > n \\ [m,\ n] & \text{se } m \leq n \end{cases}$$

- $+, -, *, /$ rappresentano le usuali operazioni aritmetiche su interi
- Il predicato $=$ rappresenta la relazione di uguaglianza
- Il predicato *in* rappresenta la relazione di appartenenza
- Il predicato *un* rappresenta la relazione di unione insiemistica definita come: $un\big(r,\, s,\, t\big) = true \iff t = r \cup s$
- Il predicato *disj* rappresenta la relazione di disgiunzione insiemistica definita come:
  $disj\big(r,\, s\big) = true \iff r \cap s = \emptyset$
- Il predicato *size* rappresenta la relazione cardinalità di insiemi definita come: $size(s,\, n) = true \iff n =\mid s \mid$
- Il predicato $\leq$ rappresenta la relazione di confronto "minore o uguale" sugli interi
- Il predicato *set* controlla che il termine sia un insieme.

2.1.3 Il solver $SAT_{\mathcal{SET}}$

La procedura $SAT_{\mathcal{SET}}$ risolve i vincoli e rappresenta il solver del linguaggio CLP$_{\mathcal{SET}}$. Il seguente algoritmo ne descrive la struttura:

```
1: procedure SAT_SET(C)
2:     C ←sort_infer(C);
3:     repeat
4:         C'← C;
5:         repeat
6:             C'' ← C;
7:             C ← STEP(C);
8:         until C = C'';
9:         C ←remove_neq(C);
10:    until C' = C;
11: return C;
```

Essenzialmente $SAT_{\mathcal{SET}}$ utilizza tre metodi:

- **sort_infer** è utilizzato per aggiungere alla $\mathcal{SET}$-*formula* C i vincoli **set** e **integer** per assicurarsi che gli oggetti utilizzati nei vincoli siano del tipo corretto.
- $STEP$ applica speciali *regole di riscrittura* alla formula C corrente e ritorna *false* oppure la formula modificata. L'esecuzione di $STEP$ viene iterata fino ad arrivare ad ottenere una formula in forma irriducibile. Notare che $STEP$ ritorna $false$ se almeno uno dei vincoli in C viene riscritto a $false$. Ovviamente anche $STEP(false)$ ritorna $false$.
- **remove_neq** tratta l'eliminazione dei $\neq -constraints$ nel caso in cui coinvolgano variabili insiemistiche che appaiono anche in vincoli di unione in forma risolta.

Quando la computazione non deterministica di $SAT_{\mathcal{SET}}(C)$ termina, se il risultato è $false$ allora si può concludere che C non è soddisfacibile; al contrario, se nessuno

dei vincoli in C viene riscritto a $false$ allora ogni soluzione generata da $SAT_{\mathcal{SET}}$ è una soluzione per C e viceversa.

2.2 Il linguaggio $\mathcal{L}_{\mathcal{BR}}$

Il linguaggio $\mathcal{L}_{\mathcal{BR}}$ estende il linguaggio CLP($\mathcal{SET}$), precedentemente decritto, con relazioni binarie. La sintassi di $\mathcal{L}_{\mathcal{BR}}$ è sostanzialmente la stessa di CLP($\mathcal{SET}$). Si richiede soltanto che l'insieme $\mathcal{F}$ venga esteso con almeno il simbolo binario $(\cdot \mid \cdot)$ che verrà usato per rappresentare le coppie. Inoltre l'insieme $\prod_C$ viene esteso con nuovi simboli di predicato su relazioni binarie, precisamente $comp, id$, inv e rel. L'insieme $\prod_C$ esteso è così descritto:

$$\prod\nolimits_C = \{ =, in, un, disj, \leq, size, set, integer \} \cup \{ comp, id, inv, rel \}.$$

Anche il solver $SAT_{\mathcal{SET}}$ viene esteso con le opportune regole di riscrittura per i nuovi vincoli $comp, id$ e inv, prendendo poi il nome di $SAT_{\mathcal{BR}}$ e, di conseguenza, anche il metodo $remove\_neq$ viene esteso applicandolo anche ai nuovi vincoli aggiunti.

2.2.1 Vincoli su relazioni binarie

Un termine insiemistico r rapperesenta una relazione binaria se assume una delle seguenti forme:

- $r = \{\}$
- $r = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- $r = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \mid S\}$ con S relazione binaria

dove:

- x_i, y_i con i= 1,...,n sono termini
- S può essere sia una variabile che un termine insiemistico.

Di seguito viene data una semantica informale dei vincoli primitivi per relazioni binarie.

- $comp\big(r,\, s,\, t\big)$: dove r, s e t sono relazioni binarie. L'interpretazione intuitiva del vincolo è

$$comp\big(r, s, t\big) = true \iff t = \big\{[x, z] \mid \exists y.[x, y] \in r \wedge [y, z] \in s\big\}$$

- $id\big(r,\, I\big)$: dove I è un insieme e r una relazione binaria che ne rappresenta l'identità. L'interpretazione intuitiva del vincolo è

$$id\big(r, I\big) = true \iff r = \big\{[x, x] \mid x \in I\big\}$$

- $inv\big(r,\, s\big)$: dove r, s sono relazioni binarie. L'interpretazione intuitiva del vincolo è

$$inv\big(r, s\big) = true \iff s = \big\{[y, x] \mid \exists [x, y] \in r\big\}$$

I vincoli *dom* e *ran* vengono realizzati come vincoli primitivi nonostante sia dimostrato che possono essere realizzati come vincoli derivati (cioè definiti tramite una formula $\mathcal{L}_{\mathcal{BR}}$). Questo viene fatto per motivi di efficienza.

- $dom\big(r,\, A\big)$: dove r è una relazione binaria e A un insieme che rappresenta il suo dominio. L'interpretazione intuitiva del vincolo è

$$dom(r, A) = true \iff A = \big\{x \mid \exists [x, y] \in r\big\}$$

- $ran\big(r,\, B\big)$: dove r è una relazione binaria e B un insieme che rappresenta il suo codominio (o rango).
  L'interpretazione intuitiva del vincolo è

$$ran\big(r, B\big) = true \iff B = \big\{y \mid \exists [x, y] \in r\big\}$$

Esempi:

- $dom\big(\{(1,2),(3,4)\},\{1,3\}\big) \implies true$
- $ran\big(\{(1,2),(3,4)\},\{2,4\}\big) \implies true$
- $comp\big(\{(1,2),(3,4)\},\{(2,6),(4,8)\},\{(1,6),(3,8)\}\big) \implies true$
- $id\big(\{(1,1),(2,2)\},\{1,2\}\big) \implies true$
- $inv\big(\{(1,2),(3,4)\},\{(2,1),(4,3)\}\big) \implies true$

I *constraint* primitivi precedentemente definiti possono essere utilizzati in congiunzione tra loro (assieme a nuove varibili ove necessita) per definire altri vincoli (*vincoli derivati*), ad esempio come segue:

$$dres\big(A,R,S\big) \iff un\big(S,N_1,R\big) \wedge dom\big(S,N_2\big) \wedge N_2 \subseteq A \wedge dom\big(N_1,N_3\big) \wedge A \parallel N_3$$
$$rres\big(R,A,S\big) \iff un\big(S,N_1,R\big) \wedge ran\big(S,N_2\big) \wedge N_2 \subseteq A \wedge ran\big(N_1,N_3\big) \wedge A \parallel N_3$$

che rappresentano rispettivamente i vincoli di dominio ristretto ($dres$) e di codominio ristretto ($rres$).

Esempi:

- $dres\big(\{1,3\},\{(1,2),(3,4),(5,6)\},\{(1,2),(3,4)\}\big) \implies true$
- $rres\big(\{(1,2),(3,4),(5,6)\},\{2,4\},\{(1,2),(3,4)\}\big) \implies true$

Anche i vincoli negativi $ndom, nran, ncomp, nin$ e $ninv$ vengono definiti in termini dei vincoli primitivi sopra citati.

2.2.2 Vincoli su funzioni parziali

Un termine insiemistico f rappresenta una funzione parziale se è una relazione binaria e vale la seguente definizione:

$$\forall x, y, y' : \big((x, y) \in f \wedge (x, y') \in f \implies y = y'\big)$$

Le funzioni parziali aggiungono il nuovo vincolo *pfun* all'insieme dei vincoli su relazioni binarie. Il vincolo *pfun* viene introdotto come vincolo primitivo nonostante anch'esso sia dimostrato essere possibile realizzarlo come vincolo derivato. Anche in questo caso si preferisce questa implementazione per motivi di efficienza.

Sintassi:

- *pfun(f)*: dove f è una relazione binaria.

L'interpretazione intuitiva del vincolo è:

- *Sia f così definito:* $f = \big\{\big(x_1, y_1\big), \big(x_2, y_2\big), \ldots, \big(x_n, y_n\big) \mid S\big\}$

  allora $pfun(f) = true \iff$
  $\forall x_{\mathrm{i}}, x_{\mathrm{j}} \in \big\{x_1, \ldots, x_{\mathrm{n}}\big\} \subset dom(f) : x_{\mathrm{i}} \neq x_{\mathrm{j}} \ \wedge \ \big\{x_1, \ldots, x_{\mathrm{n}}\big\} \notin dom(\mathrm{S})$

Quindi se *pfun(f) = true* allora f è una funzione parziale altrimenti
(*pfun(f) = false*) f è semplicemente una relazione binaria.

Notare che:
anche le funzioni parziali hanno i vincoli di *dom* e *comp* che rispettano le definizioni date in precedenza, ma hanno un diverso funzionamento e diconseguenza una diversa implementazione.
Ad esempio, se r è una relazione binaria e f una funzione parziale:

- $dom(f, \{1\}) \rightarrow f = \{(1, n)\}$
- $dom(r, \{1\}) \rightarrow r = \{(1, n_1), \cdots, (1, n_m)\}$ con $m \geq 1$

3 Implementazione del linguaggio $\mathcal{L}_{\mathcal{BR}}$ in JSetL

JSetL è una libreria Java utilizzata per il supporto della programmazione dichiarativa nel linguaggio imperativo Object Oriented Java [2].
In questo capitolo presenteremo dapprima le classi generali `LVar`, `LSet`, `LList`, `Constraint` e `SolverClass`; poi successivamente introdurremo le nuove classi `LPair`, `LRel` e `LMap` da noi sviluppate per l'implementazione del linguaggio $\mathcal{L}_{\mathcal{BR}}$ nella libreria JSetL.

3.1 Le classi del linguaggio CLP($\mathcal{SET}$)

3.1.1 La classe `LVar`

La classe `LVar` permette l'implementazione delle variabili logiche che possono contenere come loro valore qualsiasi oggetto Java (ad esempio `Integer` e `String`).
Si possono creare variabili logiche con un valore assegnato (inizializzate o bound), senza valore assegnato (non inizializzate o unbound), con o senza un nome identificativo assegnato.

Esempi:

```
/* Creazione di una variabile logica non inizializzata e senza nome*/
LVar lv1 = new LVar();
/*Creazione di una variabile logica non inizializzata di nome X*/
LVar lv2 = new LVar("X");
/*Creazione una variabile logica con valore 152 e senza nome*/
LVar lv3 = new LVar(152);
```

Di seguito vengono descritti i vincoli su `LVar`.

Siano `X` una `LVar`, `O` un `Object` e `S` un `LSet`:

- `X.eq(O)` permette di realizzare il vincolo `X = O` cha unifica la variabile `X` all'`Object O` (in particolare `O` può essere un'altra `LVar`).
- `X.neq(O)` permette di realizzare il vincolo `X` $\neq$ `O`. Il vincolo è soddisfatto se `X` e `O` sono oggetti diversi tra loro.
- `X.in(S)` realizza il vincolo atomico `X` $\in$ `S`. Quando risolto il vincolo unificherà non deterministicamente la variabile logica `X` con ogni elemento dell'insieme `S`. Il vincolo si dice risolto se almeno un'unificazione ha successo.
- `X.nin(S)` realizza il vincolo atomico `X` $\notin$ `S`. Perchè il vincolo sia vero, è richiesto che la variabile `X` non appartenga all'insieme `S`.

3.1.2 La classe `LSet`

La classe `LSet` realizza l'implementazione di insiemi logici che possono contenere qualsiasi oggetto tra cui un altro `LSet`.
Si possono creare insiemi logici inizializzati (bound), non inizializzati (unbound), con o senza un nome identificativo assegnato.
Un insieme logico può essere limitato (o chiuso) (*es.* $\{1, a, \{\}, \{b\}\}$) oppure essere parzialmente definito e quindi illimitato (o aperto)
(*es.* $\{1, a, \{\}, \{b\} \mid R\}$ con $R \in \mathcal{V}$).

Esempi:

```
/* Creazione di un insieme logico non inizializzato e senza nome*/
LSet ls1 = new LSet();
/*Creazione di un insieme logico non inizializzato di nome X*/
LSet ls2 = new LSet("X");
/*Creazione di un insieme logico limitato {1,a,{b}} con nome S*/
LSet ls3 = LSet.empty().ins(1).ins(a).ins(LSet.empty().ins(b))
              .setName("S");
/*Creazione di un insieme logico parzialmente definito
          {1,a,{b}|R} con nome S*/
```

```
LSet ls3 = new LSet().ins(1).ins(a).ins(LSet.empty().ins(b))
              .setName("S");
```

Su oggetti di tipo `LSet` si possono applicare i vincoli `eq`, `neq`, `contains`, `ncontains`, `diff`, `disj`, `inters`, `less`, `size`, `subset` e `union` che permettono di realizzare rispettivamente `R = S`, `R` $\neq$ `S`, `R` $\ni$ `X`, `X` $\notin$ `R`, `R = S` $\setminus$ `Q`, `R` $\parallel$ `S`, `R = S` $\cap$ `Q`, `R = S` $\setminus$ {`X`}, |`R`| `= i`, `R` $\subseteq$ `S`, `R = S` $\cup$ `Q`;
dove `i` è un `integer`, `X` un `LVar` e `R,S,Q` sono `LSet`.

3.1.3 La classe `Constraint`

I vincoli in JSetL sono istanze della classe `Constraint`.
La classe è un contenitore di oggetti di tipo `AConstraint` (Atomic Constraint).
La classe `AConstraint` ha i seguenti campi protected:

- `cons` contenente l'id del vincolo richiamato
- `arg1, arg2, arg3, arg4` contenenti gli oggetti su cui il vincolo richiamato lavora
- `solved`, se messo a *true*, indica che il vincolo che si stà analizzando è risolto e quindi non viene più considerato dal risolutore
- `caseControl` contenente il punto di scelta per il backtracking.

Un `Constraint` permette di mantenere in memoria un insieme di vincoli atomici.
Di seguito sono mostrati i costruttori e alcuni metodi della classe `Constraint`.

Costruttori:

- `Constraint()`: crea il vincolo vuoto
- `Constraint(String extName, Object` $\mathtt{o}_1$`, ..., Object` $\mathtt{o}_n$`)`: crea il vincolo con nome `extName` e argomenti $\mathtt{o}_1$, ..., $\mathtt{o}_n$, con $1 \leq n \leq 4$

Metodi principali:

- `Constraint and(Constraint c)`: ritorna il vincolo `this` $\wedge$ `c`
- `Constraint or(Constraint c)`: ritorna il vincolo `this` $\vee$ `c`

3.1.4 La classe `SolverClass`

In JSetL il risolutore di vincoli di CLP($\mathcal{SET}$), $SAT_{\mathcal{SET}}$, è implementato dalla classe `SolverClass`.
La classe `SolverClass` ha i seguenti campi protected:

- `storeUnchanged` utilizzato per indicare se lo store del solver è stato modificato
- `storeSize` utilizzato per conoscere la dimensione attuale dello store del risolutore
- `ConstraintStore store = new ConstraintStore(this)` rappresenta lo store del risolutore.

La classe fornisce i metodi per aggiungere vincoli, controllarne la soddisfacibilità e mostrare i vincoli presenti. I vincoli vengono risolti dal risolutore per mezzo di *regole di riscrittura*. La procedura di risoluzione termina quando tutti i vincoli sono in forma irriducibile (cioè non ci sono più regole di riscrittura da applicare), oppure viene lanciata un'eccezione di tipo `Failure`, nel caso in cui il vincolo non sia soddisfacibile.

Costruttore:

- `SolverClass()`: crea un risolutore di vincoli con `ConstraintStore` vuoto

Metodi:

- `void add(Constraint c)`: aggiunge il vincolo `c` al `ConstraintStore` del risolutore
- `void showStore()`: stampa la congiunzione di tutti i vincoli presenti nel constraint store ancora in forma non risolta
- `boolean check(Constraint c)`: controlla che i vincoli nel `ConstraintStore` del risolutore e il vincolo `c` da aggiungere siano soddisfacibili. Se possibile viene calcolata una soluzione e aggiunto il vincolo al risolutore e il metodo ritorna *true*. Se il vincolo non è soddisfacibile i vincoli nel risolutore rimangono invariati, il vincolo `c` scartato ed il metodo ritorna $false$.
- `void solve()`: risolve i vincoli nel `ConstraintStore` del risolutore. Come il metodo `check()` cerca una soluzione se possibile, altrimenti viene lanciata un'eccezione di tipo `Failure`.

3.1.5 Un primo esempio

Mostriamo ora un semplice esempio di programma Java che usa la libreria JSetL. Il programma crea un insieme chiamato `set` che contiene alcuni oggetti ed un resto `_rest` il cui contenuto è a noi sconosciuto. Il programma crea anche un solver che viene utilizzato per aggiungere il vincolo `x` $\in$ `set` e assegnare il nuovo eventuale valore a `set`. Viene poi stampato il nuovo valore di `set` ed il contenuto del solver tramite i rispettivi metodi `set.output()` e `solver.showStore()`.

```
import JSetL.*;

public class esempio1 {

  public static void main(String[] args) throws Failure{

    SolverClass solver = new SolverClass();
```

```
      LVar x = new LVar(3).setName("X");
      // set = {{15}, c, hello| _rest}
      LSet set = new LSet("rest")
            .ins("hello")
            .ins('c')
            .ins(LSet.empty().ins(15)).setName("set");
      //aggiungo un nuovo vincolo al solver
      solver.add(x.in(set));
      // mostro i vincoli presenti nello store del solver
      solver.showStore();
      /*risolvo la congiunzione di vincoli presenti nello store del solver*/
      solver.solve();
      //stampo il contenuto di set dopo la risoluzione dei vincoli
      set.output();

      /*
       * OUTPUT PROGRAMMA
       *
       * Store: 3 in {{15}, c, hello|_rest}
       * set = {{15}, c, hello, 3|_?}
       *
       */
   }

}
```

3.2 Le classi del linguaggio $\mathcal{L}_{\mathcal{BR}}$

3.2.1 La classe `LPair`

La classe `LPair` permette di realizzare la coppia logica (x,y) dove x e y sono oggetti qualunque. La classe `LPair` estende la classe `LList` che permette l'implementazione di liste logiche le quali possono contenere qualsiasi oggetto. Importante, nella realizzazione di oggetti di tipo `LList`, è l'ordine degli elementi in quanto la lista [1,a,...] $\neq$ [a,1,...] e di conseguenza questo vale anche per gli elementi di `LPair`. Si possono creare `LPair` bound o unbound, con o senza un nome identificativo.

Esempi:

```
/*Creazione di un LPair non inizializzata e con nome P1*/
LPair lp1 = new LPair("P1");
/*Creazione di un LPair inizializzata senza nome*/
LPair lp2 = new LPair(1,new LVar());
/*Creazione di un LPair inizializzata con nome P2*/
LPair lp3 = new LPair("P2",new LVar(1),5);
/*Creazione di un LPair inizializzata con un altro LPair*/
LPair lp4 = new LPair(lp2);
/*Creazione di un LPair inizializzata con un altro LPair con nome P5*/
LPair lp5 = new LPair("P5",lp3);
```

In particolare il costruttore `LPair(LPair p)` permette di creare una nuova coppia logica uguale a `p` similmente al funzionamento del metodo `this.eq(p)`.

Vincoli su LPair

- `Constraint in(LCollection s)`:
  permette di realizzare il vincolo `this` $\in$ `s`.
- `Constraint nin(LCollection s)`:
  permette di realizzare il vincolo `this` $\notin$ `s`.

Sono inoltre resi disponibili i vincoli `eq` ed `neq` che vengono ereditati dalla classe `LList`.

Metodi get

- `Object getFirst()`: ritorna il primo elemento della coppia, ovvero restituisce `x` $\iff$ `this = [x,y]`.
- `Object getSecond()`: ritorna il secondo elemento della coppia, ovvero restituisce `y` $\iff$ `this = [x,y]`.

3.2.2 La classe `LRel`

`LRel` è la classe utilizzata per la rappresentazione delle relazioni binarie in JSetL e rispetta la definizione di relazione binaria data in precedenza. Le relazioni binarie vengono quindi rappresentate come particolari insiemi i cui elementi sono coppie ordinate. La classe `LRel` estende `LSet`, dalla quale eredita in particolare i vincoli associati. In questo modo una `LRel` è un insieme logico con la particolarità che i suoi elementi sono oggetti della classe `LPair`.

Secondo la definizione precedentemente data, una relazione binaria è della forma:

- $r = \{\}$
- $r = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- $r = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \mid S\}$

I costruttori della classe `LRel` permettono e garantiscono che le relazioni generate rispettino la forma descritta sopra. In particolare:

- `public LRel();`
- `public LRel(String n, Set<LPair> s);`
- `public LRel(String n, LRel s);`

Inoltre ciò è anche garantito dall'utilizzo del metodo statico `LRel.isEmpty()` che genera un insieme vuoto, ereditato da `LSet`, e del metodo `ins(LPair p)`, che inserisce nuove coppie `LPair` nella relazione binaria che si stà creando (`this`).

Esempi:

```
/* Creazione di una relazione binaria non inizializzata e senza nome*/
LRel lr1 = new LRel();
/*Creazione di una relazione binaria non inizializzato di nome R*/
LRel lr2 = new LRel("R");
/*Creazione di una relazione binaria {[1,2], [hello,5]} con nome R*/
LRel lr3 = LRel.empty().ins(new LPair(1,2)).ins(new LPair("hello",5))
           .setName("R");
/*Creazione di una relazione binaria parzialmente definita
          {[1,a], [hello,10]|R} con nome S*/
LRel lr4 = new LRel("R").ins(new LPair(1,'a')).ins(new LPair("hello",10))
           .setName("S");
```

Vincoli

- Vincoli insiemistici: la classe `LRel`, dato che è derivata dalla classe `LSet`, eredita tutti i vincoli insiemistici da quest'ultima.
- Vincoli primitivi su relazioni binarie: sono i vincoli `dom, ran, comp, id e inv` definiti nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ e direttamente implementati nella libreria JSetL.
- Vincoli derivati su relazioni binarie: sono i vincoli ottenuti dalla congiunzione dei vincoli primitivi e sono i vincoli `dres, rres, ndom, nran, ninv, nid, ncomp, ndres, nrres`.

Generalmente un vincolo viene generato dalla classe attraverso il metodo omonimo e rispetta la seguente struttura:

```
public Constraint nomeVincolo(Class1 obj_1,..., Class_n obj_n){
    return new Constraint(this, Environment.constraintCode,
                                                    obj_1,..., obj_n);
}
```

dove `Environment.constraintCode` contiene il codice identificativo del vincolo che si stà richiamando sugli oggetti.

Vincoli primitivi della classe `LRel`

- `Constraint dom(LSet d)`: il metodo prende come argomento un oggetto `d` di tipo `LSet` che rappresenta il dominio della relazione e restituisce un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $dom(this, d)$.
- `Constraint ran(LSet r)`: il metodo prende come argomento un oggetto `r` di tipo `LSet` che rappresenta il codominio (o rango) della relazione e restituisce un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $ran(this, r)$.
- `Constraint comp(LRel s, LRel q)`: il metodo prende come argomenti due oggetti `s` e `q` di tipo `LRel` che rappresentano rispettivamente la relazione con cui bisogna effettuare la composizione e la relazione risultante dalla composizione di `this` e `s`. Il metodo restituisce poi un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $comp(this, s, q)$.
- `Constraint id(LSet a)`: il metodo prende come argomento un oggetto `a` di tipo `LSet` che rappresenta l'insieme della relazione identità e restituisce un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $id(this, a)$.
- `Constraint inv(LRel s)`: il metodo prende come argomento un oggetto `s` di tipo `LRel` che rappresenta la relazione inversa di `this` e restituisce un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $inv(this, s)$.

Di seguito viene mostrato come esempio l'implementazione del vincolo `inv`:

```
public Constraint inv(LRel s){
    return new Constraint(this,Environment.invCode,s);
}
```

Vincoli derivati della classe `LRel`

- `Constraint dres(LRel s, LSet a)`: il metodo prende come argomento un oggetto `a` di tipo `LSet` che rappresenta il dominio ristretto e un oggetto `s` di tipo `LRel` che rappresenta la relazione ristretta restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $dres(this, s, a)$.
- `Constraint rres(LRel s,LSet b)`: il metodo prende come argomento un oggetto `b` di tipo `LSet` che rappresenta il codominio ristretto e un oggetto `s` di tipo `LRel` che rappresenta la relazione ristretta restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $rres(this, s, a)$.
- `Constraint ndom(LSet a)`: il metodo prende come argomento un oggetto `a` di tipo `LSet` che rappresenta l'antidominio della relazione `this` restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $ndom(this, a)$.
- `Constraint nran(LSet a)`: il metodo prende come argomento un oggetto `a` di tipo `LSet` che rappresenta l'anticodominio della relazione `this` restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $nran(this, a)$.
- `Constraint ninv(LRel s)`: il metodo prende come argomento un oggetto `s` di tipo `LRel` che rappresenta la relazione non inversa della relazione `this` restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione

(o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $ninv(this, s)$.

- `Constraint nid(LSet a)`: il metodo prende come argomento un oggetto `a` di tipo `LSet` che rappresenta l'insieme relativo alla relazione `this` non identità restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $nid(this, a)$.

- `Constraint ncomp(LRel s,LRel t)`: il metodo prende come argomenti due oggetti `s` e `t` di tipo `LRel` dove `t` rappresenta la relazione di anticomposizione tra le relazioni `this` e `s` restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $ncomp(this, s, t)$.

- `Constraint ndres(LRel s, LSet a)`: il metodo prende come argomento un oggetto `a` di tipo `LSet` che rappresenta l'antirestrizione del dominio e un oggetto `s` di tipo `LRel` che rappresenta la relazione ristretta restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $ndres(this, s, a)$.

- `Constraint nrres(LRel s,LSet b)`: il metodo prende come argomento un oggetto `b` di tipo `LSet` che rappresenta l'antirestrizione del codominio e un oggetto `s` di tipo `LRel` che rappresenta la relazione ristretta restituendo un oggetto di tipo `Constraint` che rappresenta la congiunzione (o disgiunzione) di vincoli corrispondente alla definizione del vincolo $\mathcal{L}_{\mathcal{BR}}$ derivato $nrres(this, s, a)$.

Ogni vincolo sopra elencato genera al proprio interno dei vincoli intermedi (primitivi o anch'essi derivati), che congiunti, permettono la realizzazione del vincolo che si sta implementando.

3.2.3 La classe `LMap`

`LMap` è la classe utilizzata per la rappresentazione delle funzioni parziali in JSetL e rispetta la definizione di funzione parziale data in precedenza. Le funzioni parziali vengono quindi rappresentate come particolari relazioni binarie i cui elementi del dominio sono diversi gli uni dagli altri. La classe `LMap` estende `LRel`, dalla quale eredita in particolare i vincoli associati e ne ridefinisce alcuni per un corretto funzionamento.

La classe è fornita di costruttori che permettano di rispettare la definizione di funzione parziale data, in particolare:

- `public LMap(Set<LPair> s)`
- `public LMap(LMap m)`

Il metodo `ins(LPair p)`, grazie al metodo statico `LMap.isPfun()`, controlla che le nuove coppie che si stanno inserendo non compromettano la definizione data di funzione parziale. Sia i costruttori che il metodo `ins` generano una `NotPFunException()` nel caso in cui la definizione di funzione parziale venga violata.
Ad esempio, se `s` contiene le coppie `[1,2],[1,3]`, allora l'esecuzione di `new LMap(s)` solleva l'eccezione `NotPFunException()`.

Esempi:

```
/* Creazione di una funzione parziale non inizializzata e senza nome*/
LMap lm1 = new LMap();
/*Creazione di una funzione parziale non inizializzata di nome F*/
LMap lm2 = new LMap("F");
/*Creazione di una funzione parziale {[1,2], [10,5]} con nome F*/
LMap lm3 = LMap.empty().ins(new LPair(1,2)).ins(new LPair(new LVar(10),5))
           .setName("F");
/*Creazione di una funzione parziale parzialmente definita
          {[1,a], [x,10]|R} con nome F*/
LMap lm4 = new LMap("R").ins(new LPair(1,'a'))
           .ins(new LPair(new LVar("x"),10)).setName("F");

/*Creazione di una funzione parziale {[1,2], [1,5]} senza nome*/
```

```
LMap error = LMap.empty().ins(new LPair(1,2)).ins(new LPair(1,5)));

//in questo caso, all'inserimento della nuova coppia [1,5],
//viene lanciata una NotPFunException()
```

Il metodo statico `isPfun` viene utilizzato dai costruttori e dal metodo `ins` per verificare che non venga violata la definizione di funzione parziale. Di seguito viene mostrato il codice del metodo ausiliario `isPfun` utilizzato:

```
public static boolean isPfun(LMap p){
   LPair p1;
   LPair p2;
   for(int i = 0 ; i < p.getSize();++i){
       p1 = new LPair((LPair)p.get(i)).getEndOfEquChain();
      for(int j = i+1; j < p.getSize();++j){
         p2 = new LPair((LPair)p.get(j)).getEndOfEquChain();
         if(!p1.isInit() || !p2.isInit())
            continue;
         Object x1 = p1.getFirst();
         Object y1 = p1.getSecond();
         Object x2 = p2.getFirst();
         Object y2 = p2.getSecond();
         if(x1.equals(x2) && LObject.isGround(y1) && LObject.isGround(y2) && !y1
             .equals(y2))
            return false;
      }
   }
   return true;
}
```

dove

- `getEndOfEquChain()` ritorna il rappresentante della catena dei campi `equ`
- `LObject.isGround(..)` verifica che il rappresentante della catena dei campi `equ` sia inizializzato

- `isInit()` verifica che l'oggetto sia inizializzato

Vincoli

- Vincoli primitivi: la classe `LMap`, per come è costruita, eredita tutti i vincoli primitivi, insiemistici e relazionali, dalla classe `LRel`. Vengono ridefiniti i vincoli `dom` e `comp` in quanto ci si vuole assicurare che i vincoli vengano richiamati su oggetti di tipo `LMap` in quanto ne varia l'implementazione rispetto a `LRel` (anche il vincolo `ran` viene ridefinito ma solo per richiamare l'implementazione corretta di `comp` nelle regole di riscrittura del vincolo; pertanto non verrà mostrato tra i vincoli primitivi della classe). Viene inoltre introdotto il nuovo vincolo `pfun`.
- Vincoli derivati: la classe `LMap`, per come è costruita, eredita tutti i vincoli derivati, insiemistici e relazionali dalla classe `LRel`.

Vincoli primitivi della classe `LMap`

- `Constraint dom(LSet d)`: il metodo prende come argomento un oggetto `d` di tipo `LSet` che rappresenta il dominio della funzione parziale e restituisce un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $dom(this, d)$ (come si può notare il funzionamento è lo stesso visto in `LRel` ma applicato a `LMap`).
- `Constraint comp(LMap s,LMap q)`: il metodo prende come argomenti due oggetti `s` e `q` di tipo `LMap` che rappresentano rispettivamente la funzione con cui bisogna effettuare la composizione e la funzione risultante dalla composizione di `this` e `s`. Il metodo restituisce poi un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ $comp(this, s, q)$ (come si può notare il funzionamento è lo stesso visto in `LRel` ma applicato a `LMap`).
- `Constraint pfun()`: il metodo non ha alcun argomento come input. Il metodo restituisce poi un nuovo oggetto di tipo `Constraint` corrispondente al vincolo $\mathcal{L}_{\mathcal{BR}}$ di verifica di funzione parziale $pfun(this)$.

4 Riscrittura dei vincoli su relazioni binarie

In questo capitolo, e nel successivo, ci occuperemo di mostrare le regole di riscrittura [3] e le effettive implementazioni in JSetL dei vincoli presentati nelle classi `LRel` ed `LMap` create per implementare il linguaggio $\mathcal{L}_{\mathcal{BR}}$ all'interno della libreria Java JSetL. Più precisamente, in questo capitolo, verrà presentata in ogni sua parte la classe `RwRulesBR` contenente l'effettiva implementazione dei vincoli primitivi delle classi `LRel` ed `LMap`, mentre i vincoli derivati vengono implementati nella rispettiva classe di appartenenza. Di seguito verrano mostrate la sintassi e la notazione convenzionale utilizzata per le regole di riscrittura dei vincoli presentati.

4.1 Sintassi e notazioni convenzionali

Un $\mathcal{BR}$-constraint è un qualunque predicato basato sull'insieme di simboli di predicato $\prod$. Chiamiamo π-constraint qualsiasi letterale $\pi(x_1, \ldots, x_n)$ dove π è in simbolo in $\prod$. Una *regola di riscrittura* per $\pi \in \prod$ ha la seguente forma: $\phi \rightarrow \Phi$, dove ϕ è un π-constraint e Φ è una $\mathcal{BR}$-formula. $\mathcal{V}$ rappresenta l'insieme delle variabili e sfruttiamo la notazione $\dot{x}$, oppure $\dot{X}$, per indicare $x \in \mathcal{V}$, oppure $X \in \mathcal{V}$. N e n vengono utilizzati nella parte destra delle regole di riscrittura per rappresentare le nuove variabili create.

4.2 Il vincolo *dom* per `LRel`

Il vincolo di dominio nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ ha la forma $dom(r, a)$ dove r è una relazione binaria e a un insieme. Il vincolo risulta essere soddisfatto se e solo se l'insieme a è il dominio della relazione r.
Le regole di riscrittura per il vincolo *dom* sono le seguenti:

$$dom(\dot{R}, \dot{R}) \rightarrow \dot{R} = \emptyset \qquad (dom_1)$$

$$dom(R, \emptyset) \rightarrow R = \emptyset \qquad (dom_2)$$

$$dom(\emptyset, A) \rightarrow A = \emptyset \qquad (dom_3)$$

$$dom(\dot{R}, \{x \sqcup A\}) \rightarrow un(N_1, N_2, \dot{R}) \wedge dom(N_1, \{x\}) \wedge dom(N_2, A) \qquad (dom_4)$$

$$dom(\dot{R}, \{x\}) \rightarrow comp(\{(x, x)\}, \dot{R}, \dot{R}) \wedge \dot{R} \neq \emptyset \qquad (dom_5)$$

$$dom(\{(x, y) \sqcup R\}, A) \rightarrow A = \{x \sqcup N_1\} \wedge dom(R, N_1) \qquad (dom_6)$$

dove N_1 e N_2 sono nuovi oggetti logici non inizializzati, A è un insieme logico ed R una relazione binaria. Le regole fornite sfruttano il principio di ricorsione; le prime tre rappresentano i casi base mentre le regole $4, 5, 6$ rappresentano i casi riscorsivi. Da notare come il vincolo nella regola 5 sfrutta anche il vincolo *comp* (presentato più avanti) per la risoluzione di quel caso particolare.
$dom(\dot{R}, \dot{A})$, dove $\dot{R}$ e $\dot{A}$ sono variabili, è l'unica forma irriducibile del vincolo *dom*.

4.2.1 Implementazione in JSetL

```
private void relDomainRule(LSet a,LSet r,AConstraint s)
    throws Failure{
 manageEquChains(s);

   /** CASI BASE*/

   //Implementazione delle regole 1 e 2
   // dom(r,r) or dom(r,{})
   if(a.equals(r) || a.isInit() && a.isEmpty()){
    s.arg2 = LRel.empty();
       s.cons = Environment.eqCode;
       Solver.storeUnchanged = false;
       return;
   }
   //Implementazione della regola 3
```

```
  //dom({},a)
if(r.isInit() && r.isEmpty()){
   s.arg1 = LRel.empty();
      s.cons = Environment.eqCode;
      Solver.storeUnchanged = false;
      return;
  }

/**CASI RICORSIVI*/

//Implementazione della regola 5
//dom(r,{x}) con r varabile
if(!r.isInit() && a.isInit() && !a.isEmpty()
      && a.countAllElems()==1) {
   LRel app = LRel.empty().ins(new LPair(a.getOne(),a.getOne()));
      LPair newPair = new LPair(a.getOne(),new LVar());
      LRel N = new LRel();
      LRel newMap = N.ins(newPair);
      Solver.add(new AConstraint(app,Environment.brCompCode,N,N));
      s.arg1 = r;
      s.arg2 = newMap;
      s.cons = Environment.eqCode;
      Solver.storeUnchanged = false;
      return;
  }
//Implementazione della regola 4
//dom(r,a) con r variabile e a != {}
  if(!r.isInit() && a.isInit() && !a.isEmpty()) {
   LSet app = LSet.empty().ins(a.getOne());
      LSet ra = (LSet) a.removeOne();
      LRel N1 = new LRel();
      LRel N2 = new LRel();
      Solver.add(new AConstraint(N1,Environment.brDomCode,app));
      Solver.add(new AConstraint(N1,Environment.unionCode,N2,r));
      s.arg1 = N2;
      s.arg2 = ra;
      rel_Domain(s);
      Solver.storeUnchanged = false;
```

```
            return;
        }
        //Implementazione della regola 6
        //dom(r,a) dove r != {} e a non  specificato
        if(r.isInit()){
         LPair tmp = (LPair)r.getOne();
            LRel rest = (LRel)r.removeOne();
            LSet N1 = new LSet();
            LSet tmpDomain = N1.ins(tmp.getFirst());
            Solver.add(new AConstraint(a,Environment.eqCode,tmpDomain));
            s.arg1 = rest;
            s.arg2 = N1;
            rel_Domain(s);
            Solver.storeUnchanged = false;
            return;
        }
    }
```

Il metodo `manageEquChains(s)` modifica i campi arg1, $\cdots$, arg4 del costraint `s` per far si che ciascuno punti al rappresentante della catena di equivalenza dei campi `equ` degli oggetti.
Il metodo `getOne()` restituisce un elemento dell'insieme su cui viene invocato il metodo.
Il metodo `removeOne()` restuisce l'insieme su cui viene invocato il metodo, a cui è stato rimosso un elemento.

4.2.2 Esempi d'uso del vincolo

Tutti gli esempi che seguiranno assumono che sia presente la seguente dichiarazione:

```
SolverClass solver = new SolverClass();
```

Esempio 1

```
// creo r1 = {(1,2),(3,4)} con nome "r1"
LRel r1 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4))
                                        .setName("r1");
// creo d1 LSet non inizializzato con nome "d1"
LSet d1 = new LSet("d1");
solver.add(r1.dom(d1));
solver.solve();
solver.showStore();
r1.output();
d1.output();

/* OUTPUT
   Store: []            // store del solver vuoto
   r1 = {[3,4],[1,2]}
   d1 = {3,1}
 */
```

Esempio 2

```
// creo r2 = {(1,2),(3,4)| R} con nome "r2"
LRel r2 = new LRel("R").ins(new LPair(1,2)).ins(new LPair(3,4))
                                     .setName("r2");
// creo d2 = {1,5| D} con nome "d2"
LSet d2 = new LSet("D").ins(1).ins(5).setName("d2");
solver.add(r2.dom(d2));
solver.solve();
solver.showStore();
r2.output();
d2.output();

/* OUTPUT
   Store: comp({[5,5]},_N127,_N127)
   r2 = {[3,4],[1,2],[5,_N124]|_N127}
   d2 = {5,1,3|_N71}
*/
```

Esempio 3

```
// creo r3 LRel non inizializzato con nome "r3"
LRel r3 = new LRel("r3");
// creo d3 LSet non inizializzato con nome "d3"
LSet d3 = new LSet("d3");
solver.add(r3.dom(d3));
solver.solve();
solver.showStore();
r3.output();
d3.output();

/*OUTPUT
   Store: mapDom(_r3,_d3) // mantengo il vincolo dato che  forma risolta
   r3 = unknown
   d3 = unknown
 */
```

Esempio 4

```
// creo r4 = {(1,2),(3,4)} con nome "r4"
LRel r4 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4)).setName("r4
    ");
// creo d4 = {1,5} con nome "d4"
LSet d4 = LSet.empty().ins(1).ins(5).setName("d4");
solver.add(r4.dom(d4));
solver.solve();
solver.showStore();
r4.output();
d4.output();

/* OUTPUT
   JSetL.Failure
*/
```

4.3 Il vincolo *ran* per `LRel`

Il vincolo di codominio nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ ha la forma $ran(r, a)$ dove r è una relazione binaria e a un insieme.
Il vincolo risulta essere soddisfatto se e solo se l'insieme a è il codominio della relazione r. Il vincolo ha le stesse regole di riscrittura sia per relazioni binarie generiche che per funzioni parziali, di conseguenza verranno mostrate una sola volta assieme alla loro implementazione.
Le regole di riscrittura per il vincolo *ran* sono le seguenti:

$$ran(\dot{R}, \dot{R}) \rightarrow \dot{R} = \emptyset \qquad (ran_1)$$

$$ran(R, \emptyset) \rightarrow R = \emptyset \qquad (ran_2)$$

$$ran(\emptyset, A) \rightarrow A = \emptyset \qquad (ran_3)$$

$$ran(\dot{R}, \{y \sqcup A\}) \rightarrow un(N_1, N_2, \dot{R}) \wedge ran(N_1, \{y\}) \wedge ran(N_2, A) \qquad (ran_4)$$

$$ran(\dot{R}, \{y\}) \rightarrow comp(\dot{R}, \{(y, y)\}, \dot{R}) \wedge \dot{R} \neq \emptyset \qquad (ran_5)$$

$$ran(\{(x, y) \sqcup R\}, A) \rightarrow A = \{y \sqcup N_1\} \wedge ran(R, N_1) \qquad (ran_6)$$

dove N_1 e N_2 sono nuovi oggetti logici non inizializzati, A è un insieme logico ed R una relazione binaria. Le Regole fornite sfruttano il principio di ricorsione; le prime tre rappresentano i casi base mentre le regole $4, 5, 6$ rappresentano i casi riscorsivi. Da notare come il vincolo nella regola 5 sfrutta anche il vincolo *comp* (presentato più avanti) per la risoluzione di quel caso particolare. Si può inoltre notare come le regole di riscrittura del vincolo *ran* siano molto simili a quelle date per il vincolo *dom* su `LRel` con la differenza che si opera sul secondo elemento delle coppie che formano la relazione R.
$ran(\dot{R}, \dot{A})$, dove $\dot{R}$ e $\dot{A}$ sono variabili, è l'unica forma irriducibile del vincolo *ran*.

4.3.1 Implementazione in JSetL

```
private void mapRangeRule(LSet a,LSet r,AConstraint s)
     throws Failure{
  manageEquChains(s);

    /**CASI BASE*/

    //Implementazione delle regole 1 e 2
    // ran(r,r) or ran(r,{})
    if(a.equals(r) || a.isInit() && a.isEmpty()){
     s.arg2 = LRel.empty();
        s.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }
    //Iplementazione della regola 3
    //ran({},a)
    if(r.isInit() && r.isEmpty()){
     s.arg1 = LRel.empty();
        s.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }

    /**CASI RICORSIVI*/

    //Iplementazione della regola 5
    //ran(r,{y}) dove r  una variabile
    if(!r.isInit() && a.isInit() && !a.isEmpty() &&
        a.countAllElems()==1 && a.getTail().isInit()) {
     LPair newPair = new LPair(new LVar(),a.getOne());
     LRel app;
        LRel N;
        LRel newMap;
        // in base al tipo dell'oggetto su cui st lavorando
        // vengono creati gli oggetti del tipo corretto
        // in modo tale da utilizzare poi il vincolo comp corretto
```

```
    if(s.cons == Environment.brRanCode) {
       app = LRel.empty().ins(new LPair(a.getOne(),a.getOne()));
       N = new LRel();
       newMap = N.ins(newPair);
       Solver.add(new AConstraint(N,Environment.brCompCode,app,N));
    }
    else {
       app = LMap.empty().ins(new LPair(a.getOne(),a.getOne()));
       N = new LMap();
       newMap = N.ins(newPair);
       Solver.add(new AConstraint(N,Environment.pfCompCode,app,N));
    }
    s.arg1 = r;
    s.arg2 = newMap;
    s.cons = Environment.eqCode;
    Solver.storeUnchanged = false;
    return;
}
//Iplementazione della regola 4
//ran(r,a) dove r  una variabile e a != {}
if(!r.isInit() && a.isInit() && !a.isEmpty()) {
 LSet app = LSet.empty().ins(a.getOne());
    LSet ra = (LSet) a.removeOne();
    LRel N1 = new LRel();
    LRel N2 = new LRel();
    Solver.add(new AConstraint(N1,Environment.pfRanCode,app));
    Solver.add(new AConstraint(N1,Environment.unionCode,N2,r));
    s.arg1 = N2;
    s.arg2 = ra;
    range(s);
    Solver.storeUnchanged = false;
    return;
}
//Iplementazione della regola 6
//ran(r,a) dove r != {} e a non  specificata
if(r instanceof LRel && r.isInit()){
 LPair tmp = (LPair)r.getOne();
    LSet rest = r.removeOne();
```

```
        LSet N1 = new LSet();
        LSet tmpRange = N1.ins(tmp.getSecond());
        Solver.add(new AConstraint(a,Environment.eqCode,tmpRange));
        s.arg1 = rest;
        s.arg2 = N1;
        range(s);
        Solver.storeUnchanged = false;
        return;
    }
}
```

4.3.2 Esempi d'uso del vincolo

Esempio 1

```
// creo r1 = {(1,2),(3,4)} con nome "r1"
LRel r1 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4))
                                        .setName("r1");
// creo ran_1 LSet non inizializzato con nome "ran_1"
LSet ran_1 = new LSet("ran_1");
solver.add(r1.ran(ran_1));
solver.solve();
solver.showStore();
r1.output();
ran_1.output();

/* OUTPUT
   Store: []           // store del solver vuoto
   r1 = {[3,4],[1,2]}
   ran_1 = {4,2}
 */
```

Esempio 2

```
// creo r2 = {(1,2),(3,4)| R} con nome "r2"
LRel r2 = new LRel("R").ins(new LPair(1,2)).ins(new LPair(3,4))
                                        .setName("r2");
// creo ran_2 = {1,5| D} con nome "ran_2"
LSet ran_2 = new LSet("D").ins(1).ins(5).setName("ran_2");
solver.add(r2.ran(ran_2));
solver.solve();
solver.showStore();
r2.output();
ran_2.output();

/* OUTPUT
   Store: ran(_N140,_N118) AND [_N142,5] nin _N329 AND . . .
   r2 = {[3,4],[1,2],[_N142,5],[_N155,1]|_N329}
   ran_2 = {5,1,4,2|_N118}
*/
```

Esempio 3

```
// creo r3 LRel non inizializzato con nome "r3"
LRel r3 = new LRel("r3");
// creo ran_3 LSet non inizializzato con nome "ran_3"
LSet ran_3 = new LSet("ran_3");
solver.add(r3.ran(ran_3));
solver.solve();
solver.showStore();
r3.output();
ran_3.output();

/*OUTPUT
   Store: mapDom(_r3,_ran_3) // mantengo il vincolo dato che  forma
       risolta
   r3 = unknown
   ran_3 = unknown
 */
```

Esempio 4

```
// creo r4 = {(1,2),(3,4)} con nome "r4"
LRel r4 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4)).setName("r4
    ");
// creo ran_4 = {1,5} con nome "ran_4"
LSet ran_4 = LSet.empty().ins(1).ins(5).setName("ran_4");
solver.add(r4.ran(ran_4));
solver.solve();
solver.showStore();
r4.output();
ran_4.output();

/* OUTPUT:  JSetL.Failure */
```

4.4 Il vincolo *comp* per LRel

Il vincolo di composizione sequenziale nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ ha la forma $comp(r, s, t)$ dove r, s, t sono relazioni binarie. Il vincolo risulta essere soddisfatto se e solo se la relazione t è la composizione sequenziale delle relazioni r ed s ($t = r \circ s$).
Le regole di riscrittura per il vincolo *comp* sono le seguenti:

$$comp(\emptyset, S, T) \rightarrow T = \emptyset \qquad (\circ_1)$$

$$comp(R, \emptyset, T) \rightarrow T = \emptyset \qquad (\circ_2)$$

$$\begin{aligned} comp(\{(x,u)\},\{(t,z)\},T) \rightarrow \\ u = t \wedge T = \{(x,z)\} \bigvee u \neq t \wedge T = \emptyset \end{aligned} \qquad (\circ_3)$$

$$\begin{aligned} comp(\{(x,u) \sqcup R\},\{(t,z) \sqcup S\},\emptyset) \rightarrow \\ u \neq t \wedge comp(\{(x,u)\}, S, \emptyset) \\ \wedge comp(R, \{(t,z)\}, \emptyset) \wedge comp(R, S, \emptyset) \end{aligned} \qquad (\circ_4)$$

$$comp(\{(x,x)\}, \{(x,y) \sqcup S\}, \{(x,y) \sqcup S\}) \rightarrow comp(\{(x,x)\}, S, S) \qquad (\circ_{4.1})$$

$$comp(\{(x,y) \sqcup R\}, \{(y,y)\}, \{(x,y) \sqcup R\}) \rightarrow comp(R, \{(x,x)\}, R) \qquad (\circ_4)$$

$$\begin{aligned} comp(\{(x,u) \sqcup R\},\{(t,z) \sqcup S\},\dot{T}) \rightarrow \\ comp(\{(x,u)\}, \{(t,z)\}, N_1) \wedge comp(\{(x,t)\}, S, N2) \\ \wedge comp(R, \{(u,z)\}, N_3) \\ \wedge comp(R, S, N_4) \wedge un(N_1, N_2, N_3, N_4, \dot{T}) \end{aligned} \qquad (\circ_5)$$

$$
\begin{aligned}
comp(R,S,\{(x,z) \sqcup T\}) \rightarrow \\
& un(N_x, N_{rt}, R) \wedge un(N_z, N_{st}, S) \\
& \wedge N_x = \{(x,n) \sqcup N_1\} \wedge N_z = \{(n,z) \sqcup N_2\} \\
& \wedge comp(\{(x,x)\}, N_1, N_1) \wedge comp(N_2, \{(z,z)\}, N_2) \\
& \wedge comp(N_x, N_{st}, N_3) \wedge comp(N_{rt}, N_z, N_4) \\
& \wedge comp(N_{rt}, N_{st}, N_5) \wedge un(N_3, N_4, N_5, T)
\end{aligned}
\qquad (\circ_6)
$$

dove $N_1, N_2, N_3, N_4, N_5, N_x, N_z, N_{st}$ ed N_{rt} sono nuovi oggetti logici non inizializzati, R, S e T sono relazioni binarie. Le regole fornite sfruttano il principio di ricorsione; le prime tre rappresentano i casi base mentre le regole $4, 4.1, 4.2, 5, 6$ rappresentano i casi riscorsivi.

Forme irriducibili del vincolo:

- $comp(\dot{R}, S, \dot{T})$ con S $\neq \emptyset$
- $comp(R, \dot{S}, \dot{T})$ con R $\neq \emptyset$
- $comp(\dot{R}, S, \emptyset)$
- $comp(R, \dot{S}, \emptyset)$

dove $\dot{R}, \dot{S}, \dot{T}$ sono variabili.

4.4.1 Implementazione in JSetL

```
private void compRule_LRel(LSet r,LSet s, LSet q, AConstraint c)
        throws Failure{
     manageEquChains(c);

       /**CASI BASE*/

       /**CASI RICORSIVI*/
}
```

Vediamo prima l'implementazione dei casi base poi dei casi ricorsivi

```
       //Implementazione della regola 1
       //comp({},s,q)
       if(r.isInit() && r.isEmpty()){
            Solver.add(new AConstraint(q,Environment.eqCode,LRel.empty()));
            c.solved=true;
            return;
        }
       //Implementazione della regola 2
       //comp(r,{},q)
        if(s.isInit() && s.isEmpty()){
         Solver.add(new AConstraint(q,Environment.eqCode,LRel.empty()));
            c.solved=true;
            return;
        }

        //Implementazione della regola 3
        //comp({(x,y)},{(y,z)},q)
        if(r.isInit() && s.isInit() && !r.isEmpty() && !s.isEmpty() &&
            r.countAllElems() == 1 && s.countAllElems() == 1 &&
            r.getTail().isInit()
            && s.getTail().isInit()) {
         LPair Rpair = (LPair)r.getOne();
         LPair Spair = (LPair)s.getOne();
         switch (c.caseControl) {
            case 0:
```

```
            Solver.addChoicePoint(c);
            LRel rel = LRel.empty().ins(new LPair(Rpair.getFirst(),Spair.
                getSecond()));
            Solver.add(new AConstraint(Rpair.getSecond(),Environment.eqCode,
                Spair.getFirst()));
            Solver.add(new AConstraint(q,Environment.eqCode,rel));
            c.solved = true;
            return;
        case 1:
           LRel emptyRel = LRel.empty();
           Solver.add(new AConstraint(Rpair.getSecond(),Environment.neqCode,
               Spair.getFirst()));
           Solver.add(new AConstraint(q,Environment.eqCode,emptyRel));
           c.solved = true;
           return;
     }
    }
```

Il metodo `getTail()` restituisce la coda dell'insieme su cui viene richiamato, cioè $\emptyset$ o un oggetto di tipo `LVar` non inizializzato.
Il metodo `addChoicePoint()` aggiunge un nuovo punto di scelta al meccanismo di backtracking.

Vediamo ora l'implementazione dei casi ricorsivi del vincolo

```
//Implementazione della regola 4
//comp(r,s,{}) dove r ed s != {}
if(q.isInit() && q.isEmpty() && r.isInit() && s.isInit()) {
 LPair Rpair = (LPair)r.getOne();
 LPair Spair = (LPair)s.getOne();
 LSet resR = r.removeOne();
 LSet resS = s.removeOne();
 LRel appR = LRel.empty().ins(Rpair);
 LRel appS = LRel.empty().ins(Spair);
 LRel emptyLRel = LRel.empty();
 Solver.add(new AConstraint(Rpair.getSecond(),Environment.neqCode,
     Spair.getFirst()));
```

```
  Solver.add(new AConstraint(appR,Environment.brCompCode,resS,emptyLRel)
      );
  Solver.add(new AConstraint(resR,Environment.brCompCode,appS,emptyLRel)
      );
  c.arg1 = resR;
  c.arg2 = resS;
  c.arg3 = emptyLRel;
  rel_comp(c);
  Solver.storeUnchanged = false;
  return;
 }
 //Implementazione della regola 4.1
 //comp({(x,x)},{(x,y)|T},{(x,y)|T})
 if(r.isInit() && !r.isEmpty() &&
       s.isInit() && !s.isEmpty() &&
       q.isInit() && !q.isEmpty() &&
       s.removeOne().equals(q.removeOne())
       && r.removeOne().isEmpty()){
    LPair rPair = (LPair) r.getOne();
    LPair sPair = (LPair) s.getOne();
    LPair qPair = (LPair) q.getOne();
    if(rPair.getFirst().equals(rPair.getSecond()) &&
           rPair.getFirst().equals(sPair.getFirst()) &&
           rPair.getFirst().equals(qPair.getFirst()) &&
           sPair.getSecond().equals(qPair.getSecond())){
       c.arg2 = s.removeOne();
       c.arg3 = q.removeOne();
       c.caseControl = 0;
       Solver.storeUnchanged = false;
       return;
    }
}
//Implementazione della regola 5
//comp(r,s,q) dove q  variabile e r,s != {}
 if(!q.isInit() && r.isInit() && s.isInit() ) {
  LPair Rpair = (LPair)r.getOne();
  LPair Spair = (LPair)s.getOne();
  LSet resR = r.removeOne();
```

```
 LSet resS = s.removeOne();
 LRel appR = LRel.empty().ins(Rpair);
 LRel appS = LRel.empty().ins(Spair);
 LRel N1 = new LRel();
 LRel N2 = new LRel();
 LRel N3 = new LRel();
 LRel N4 = new LRel();
 LRel Nunion1 = new LRel();
 LRel Nunion2 = new LRel();
 Solver.add(new AConstraint(appR,Environment.brCompCode,appS,N1));
 Solver.add(new AConstraint(appR,Environment.brCompCode,resS,N2));
 Solver.add(new AConstraint(resR,Environment.brCompCode,appS,N3));
 Solver.add(new AConstraint(N1,Environment.unionCode,N2,Nunion1));
 Solver.add(new AConstraint(Nunion1,Environment.unionCode,N3,Nunion2));
 Solver.add(new AConstraint(Nunion2,Environment.unionCode,N4,q));
 c.arg1 = resR;
 c.arg2 = resS;
 c.arg3 = N4;
 rel_comp(c);
 Solver.storeUnchanged = false;
 return;
}
//Implementazione della regola 6
//comp(r,s,q) dove q != {} ed r,s non sono specificate
if(q.isInit() && !q.isEmpty()) {
 LPair Qpair = (LPair)q.getOne();
 LSet resQ = (LSet)q.removeOne();
 LRel N1 = new LRel();
 LRel N2 = new LRel();
 LRel N3 = new LRel();
 LRel N4 = new LRel();
 LRel N5 = new LRel();
 LRel Nx = new LRel();
 LRel Nz = new LRel();
 LRel Nst = new LRel();
 LRel Nrt = new LRel();
 LVar n = new LVar();
 LPair app1 = new LPair(Qpair.getFirst(),n);
```

```
        LPair app2 = new LPair(n,Qpair.getSecond());
        LPair appxx = new LPair(Qpair.getFirst(),Qpair.getFirst());
        LPair appzz = new LPair(Qpair.getSecond(),Qpair.getSecond());
        LRel ZZ = LRel.empty().ins(appzz);
        LRel XX = LRel.empty().ins(appxx);
        LRel appNx = N1.ins(app1);
        Solver.add(N1.ncontains(app1));
        LRel appNz = N2.ins(app2);
        Solver.add(N2.ncontains(app2));
        LRel Nunion1 = new LRel();
        Solver.add(new AConstraint(Nx,Environment.unionCode,Nrt,r));
        Solver.add(new AConstraint(Nz,Environment.unionCode,Nst,s));
        Solver.add(new AConstraint(Nx,Environment.eqCode,appNx));
        Solver.add(new AConstraint(Nz,Environment.eqCode,appNz));
        Solver.add(new AConstraint(XX,Environment.brCompCode,N1,N1));
        Solver.add(new AConstraint(N2,Environment.brCompCode,ZZ,N2));
        Solver.add(new AConstraint(Nx,Environment.brCompCode,Nst,N3));
        Solver.add(new AConstraint(Nrt,Environment.brCompCode,Nz,N4));
        Solver.add(new AConstraint(N3,Environment.unionCode,N4,Nunion1));
        Solver.add(new AConstraint(Nunion1,Environment.unionCode,N5,resQ));
        c.arg1 = Nrt;
        c.arg2 = Nst;
        c.arg3 = N5;
        rel_comp(c);
        Solver.storeUnchanged = false;
        return;
       }
   }
```

La regola 4.2 del vincolo *comp* è stata omessa nella rappresentazione in quanto è analoga alla regola 4.1 con la differenza che `r` viene sostituito da s.

4.4.2 Esempi d'uso del vincolo

Esempio 1

```
// creo r1 LRel non inizializzato con nome "r1"
LRel r1 = new LRel("r1");
// creo s1 LRel non inizializzato con nome "s1"
LRel s1 = new LRel("s1");
// creo q1 = {(1,6)} con nome "q1"
LRel q1 = new LRel().ins(new LPair(1,6)).setName("q1");
solver.add(r1.comp(s1, q1));
solver.solve();
solver.showStore();
r1.output();
s1.output();
q1.output();

/*OUTPUT
   Store: comp(_N14,_N13,_N10) AND [1,_N15] nin _N6 AND . . .
   r1 = {[1,_N15]|_N36}
   s1 = {[_N15,6]|_N60}
   q1 = {[1,6]|_N1}

 */
```

Esempio 2

```
// creo r3 = {(1,6)} con nome "r2"
LRel r2 = new LRel().ins(new LPair(1,2)).setName("r2");
// creo s2 = {(2,6)} con nome "s2"
LRel s2 = new LRel().ins(new LPair(2,6)).setName("s2");
// creo q2 LRel non inizializzato con nome "q2"
LRel q2 = new LRel("q2");
solver.add(r2.comp(s2, q2));
solver.solve();
solver.showStore();
r2.output();
s2.output();
q2.output();
```

```
/*OUTPUT
   Store: comp(_N114,_N118,_N133) AND [1,6] nin _N133 AND . . .
   r2 = {[1,2]|_N114}
   s2 = {[2,6]|_N118}
   q2 = {[1,6]|_N180}
 */
```

Esempio 3

```
// creo r3 LRel non inizializzato con nome "r3"
LRel r3 = new LRel("r3");
// creo s3 = {(2,6),(4,8)} con nome "s3"
LRel s3 = LRel.empty().ins(new LPair(2,6)).
           ins(new LPair(4,8)).setName("s3");

// creo q3 LRel non inizializzato con nome "q3"
LRel q3 = new LRel("q3");
solver.add(r3.comp(s3, q3));
solver.solve();
solver.showStore();
r3.output();
s3.output();
q3.output();

/*OUTPUT
   Store: comp(_r3,{[4,8],[2,6]},_q3) // forma risolta
   r3 = unknown
   s3 = {[4,8],[2,6]}
   q3 = unknown
 */
```

Esempio 4

```
// creo r4 = {(1,2),(3,4)} con nome "r4"
LRel r4 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4));
// creo s4 = {(2,6),(4,8)} con nome "s4"
LRel s4 = LRel.empty().ins(new LPair(2,6)).ins(new LPair(4,8));
// creo q4 = {}
LRel q4 = LRel.empty();
solver.add(r4.comp(s4, q4));
solver.solve();
solver.showStore();
r4.output();
s4.output();
q4.output();

/*OUTPUT  JSetL.Failure */
```

4.5 Il vincolo *id* per `LRel`

Il vincolo di relazione identità nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ ha la forma $id(a, r)$, dove r è una relazione binaria e a un insieme. Il vincolo risulta essere soddisfatto se e solo se r è relazione identità dell'insieme a. Per comodità, nell'implementazione del vincolo, questo viene rappresentato come $\underline{id(r, a)}$ e non come descritto sopra.
Le regole di riscrittura per il vincolo id sono le seguenti:

$$id(\dot{R}, \dot{R}) \rightarrow \dot{R} = \emptyset \qquad (id_1)$$

$$id(\emptyset, R) \rightarrow R = \emptyset \qquad (id_2)$$

$$id(A, \emptyset) \rightarrow A = \emptyset \qquad (id_3)$$

$$id(\{x \sqcup A\}, R) \rightarrow R = \{(x, x) \sqcup N\} \wedge id(A, N) \qquad (id_4)$$

$$id(A, \{(x, y) \sqcup R\}) \rightarrow x = y \wedge A = \{x \sqcup N\} \wedge id(N, R) \qquad (id_5)$$

dove N è un nuovo oggetto logico non inizializzato, A è un insieme logico ed R una relazione binaria. Le regole fornite sfruttano il principio di ricorsione; le prime tre rappresentano i casi base mentre le regole 4 e 5 rappresentano i casi riscorsivi. $id(\dot{R}, \dot{A})$, dove $\dot{R}$ e $\dot{A}$ sono variabili, è l'unica forma irriducibile del vincolo id.

4.5.1 Implementazione in JSetL

```
private void idRule(LSet r, LSet a, AConstraint c)
     throws Failure {
    manageEquChains(c);

    /**CASI BASE*/

    //Implementazione della regola 1
    //id(r,r) dove r  una variabile
    if(!r.isInit() && r.equals(a)) {
     c.arg2 = LRel.empty();
        c.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }
    //Implementazione della regola 2
    //id(r,{})
    if(a.isInit() && a.isEmpty()) {
     c.arg2 = LRel.empty();
        c.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }
    //Implementazione della regola 3
    //id({},a)
    if(r.isInit() && r.isEmpty()) {
     c.arg1 = LRel.empty();
        c.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }
```

```
        /**CASI RISCORSIVI*/

        //Implementazione della regola 5
        //id(r,a) con r != {} e a non specificato
        if(r.isInit() && !r.isEmpty()) {
            LPair hr = (LPair)r.getOne();
            LRel rr = (LRel)r.removeOne();
            LSet appSet = new LSet().ins(hr.getFirst());
            Solver.add(new AConstraint(hr.getFirst(),Environment.eqCode,hr.
                getSecond()));
            Solver.add(new AConstraint(a,Environment.eqCode,appSet));
            c.arg1 = rr;
            c.arg2 = appSet.removeOne();
            id(c);
            Solver.storeUnchanged = false;
            return;
        }
        //Implementazione della regola 4
        //id(r,a) con a != {} ed r non specificato
        if(a.isInit() && !a.isEmpty()) {
            Object ha = (Object)a.getOne();
            LSet ra = (LSet)a.removeOne();
            LPair appPair = new LPair(ha,ha);
            LRel appMap = new LRel().ins(appPair);
            Solver.add(new AConstraint(r,Environment.eqCode,appMap));
            c.arg1 = appMap.removeOne();
            c.arg2 = ra;
            id(c);
            Solver.storeUnchanged = false;
            return;
        }
    }
```

4.5.2 Esempi d'uso del vincolo

Esempio 1

```
// creo r1 LRel non inizializzato con nome "r1"
LRel r1 = new LRel("r1");
// creo id_1 = {5,6} con nome "id_1"
LSet id_1 = LSet.empty().ins(5).ins(6).setName("id_1");
solver.add(r1.id(id_1));
solver.solve();
solver.showStore();
r1.output();
id_1.output();

/* OUTPUT
   Store: []           // store del solver vuoto
   r1 = {[6,6],[5,5]}
   id_1 = {6,5}
 */
```

Esempio 2

```
// creo r2 = {(2,2),(3,3)| R} di nome "r2"
LRel r2 = new LRel().ins(new LPair(2,2)).ins(new LPair(3,3))
                                    .setName("r2");
// creo id_2 = {1,5| D} con nome "id_2"
LSet id_2 = new LSet().ins(1).ins(5).setName("id_2");
solver.add(r2.id(id_2));
solver.solve();
solver.showStore();
r2.output();
id_2.output();

/* OUTPUT
   Store: _N138 id _N119
   r2 = {[3,3],[2,2],[5,5],[1,1]|_N138}
   id_2 = {5,1,3,2|_N119}
*/
```

Esempio 3

```
// creo r3 LRel non inizializzato con nome "r3"
LRel r3 = new LRel("r3");
// creo d3 LSet non inizializzato con nome "d3"
LSet id_3 = new LSet("id_3");
solver.add(r3.id(id_3));
solver.solve();
solver.showStore();
r3.output();
id_3.output();

/*OUTPUT
   Store: _r3 id _id_3 // mantengo il vincolo dato che  forma risolta
   r3 = unknown
   id_3 = unknown
 */
```

Esempio 4

```
// creo r4 = {(2,2),(4,4)} con nome "r4"
LRel r4 = LRel.empty().ins(new LPair(2,2)).ins(new LPair(4,4))
                                        .setName("r4");
// creo id_4 = {1,5} con nome "id_4"
LSet id_4 = LSet.empty().ins(1).ins(5).setName("id_4");
solver.add(r4.id(id_4));
solver.solve();
solver.showStore();
r4.output();
id_4.output();

/* OUTPUT:  JSetL.Failure */
```

4.6 Il vincolo *inv* per `LRel`

Il vincolo di relazione inversa nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ ha la forma $inv(r, s)$ dove r ed s sono relazioni binarie. Il vincolo risulta essere soddisfatto se e solo se s è relazione inversa della relazione r. Le regole di riscrittura per il vincolo inv sono le seguenti:

$$inv(R, \emptyset) \rightarrow R = \emptyset \qquad (1^{\smile})$$

$$inv(\emptyset, S) \rightarrow S = \emptyset \qquad (2^{\smile})$$

$$inv(R, \{(x, y) \sqcup S\}) \rightarrow R = \{(x, y) \sqcup N\} \wedge inv(N, S) \qquad (3^{\smile})$$

$$inv(\{(x, y) \sqcup R\}, S) \rightarrow S = \{(y, x) \sqcup N\} \wedge inv(R, N) \qquad (4^{\smile})$$

dove N è un nuovo oggetto logico non inizializzato, R ed S sono relazioni binarie. Le regole fornite sfruttano il principio di ricorsione; le prime due rappresentano i casi base mentre le regole 3 e 4 rappresentano i casi riscorsivi.
$inv(\dot{R}, \dot{S})$, dove $\dot{R}$ e $\dot{S}$ sono variabili, è l'unica forma irriducibile del vincolo inv.

4.6.1 Implementazione in JSetL

```
private void invRule(LSet r, LSet s, AConstraint c)
    throws Failure {
  manageEquChains(c);

    /**CASI BASE*/

    //Implementazione della regola 1
    //inv(r,{})
    if(s.init && s.isEmpty()) {
        c.arg2 = LRel.empty();
        c.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }
```

```
//Implementazione della regola 2
//inv({},s)
if(r.isInit() && r.isEmpty()) {
 c.arg1 = LRel.empty();
    c.cons = Environment.eqCode;
    Solver.storeUnchanged = false;
    return;
}

/**CASI RICORSIVI*/

//Implementazione della regola 4
//inv(r,s) con r != {} ed s non specificata
if(r.isInit() && !r.isEmpty()) {
 LPair hr = (LPair) r.getOne();
    LRel rr = (LRel) r.removeOne();
    LPair pairApp = new LPair(hr.getSecond(),hr.getFirst());
    LRel mapApp = new LRel().ins(pairApp);
    Solver.add(new AConstraint(s,Environment.eqCode,mapApp));
    c.arg1 = rr;
    c.arg2 = mapApp.removeOne();
    inv(c);
    Solver.storeUnchanged = false;
    return;
}
//Implementazione della regola 3
//inv(r,s) s != {} ed r non specificata
if(s.isInit() && !s.isEmpty()) {
 LPair hs = (LPair) s.getOne();
    LSet rs = s.removeOne();
    LPair pairApp = new LPair(hs.getSecond(),hs.getFirst());
    LRel mapApp = new LRel().ins(pairApp);
    Solver.add(new AConstraint(r,Environment.eqCode,mapApp));
    c.arg1 = mapApp.removeOne();
    c.arg2 = rs;
    inv(c);
    Solver.storeUnchanged = false;
    return;
```

```
    }
}
```

4.6.2 Esempi d'uso del vincolo

Esempio 1

```
// creo r1 = {(1,2),(3,4)} con nome "r1"
LRel r1 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4))
                                        .setName("r1");
// creo inv_1 LSet non inizializzato con nome "inv_1"
LRel inv_1 = new LRel("inv_1");
solver.add(r1.inv(inv_1));
solver.solve();
solver.showStore();
r1.output();
inv_1.output();

/* OUTPUT
   Store: []             // store del solver vuoto
   r1 = {[3,4],[1,2]}
   inv_1 = {[4,3],[2,1]}
 */
```

Esempio 2

```
// creo r2 = {(1,2),(3,4)| R} con nome "r2"
LRel r2 = new LRel("R").ins(new LPair(1,2)).ins(new LPair(3,4))
                                        .setName("r2");

// creo inv_2 = {(5,6),(7,8)| D} con nome "inv_2"
LRel inv_2 = new LRel("D").ins(new LPair(5,6)).ins(new LPair(7,8))
                                        .setName("inv_2");

solver.add(r2.inv(inv_2));
solver.solve();
solver.showStore();
```

```
r2.output();
inv_2.output();

/* OUTPUT
   Store: _N185 inv _N166
   r2 = {[3,4],[1,2],[8,7],[6,5]|_N185}
   inv_2 = {[7,8],[5,6],[4,3],[2,1]|_N166}
*/
```

Esempio 3

```
// creo r3 LRel non inizializzato con nome "r3"
LRel r3 = new LRel("r3");
// creo inv_3 LRel non inizializzato con nome "inv_3"
LRel inv_3 = new LRel("inv_3");
solver.add(r3.inv(inv_3));
solver.solve();
solver.showStore();
r3.output();
inv_3.output();

/*OUTPUT
   Store: _r3 inv _inv_3 // mantengo il vincolo dato che  forma risolta
   r3 = unknown
   inv_3 = unknown
 */
```

Esempio 4

```
// creo r4 = {(1,2),(3,4)} con nome "r4"
LRel r4 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(3,4))
                                    .setName("r4");
// creo inv_4 = {(1,2),(4,3)} con nome "inv_4"
LRel inv_4 = LRel.empty().ins(new LPair(1,2)).ins(new LPair(4,3))
                                    .setName("inv_4");
solver.add(r4.inv(inv_4));
solver.solve();
solver.showStore();
r4.output();
```

```
inv_4.output();

/* OUTPUT:  JSetL.Failure */
```

4.7 Vincoli derivati su `LRel`

Di seguito verranno presentati i vincoli derivati creati a partire dai vincoli già esistenti $dom, ran, un, disj$ e $subset$. Verranno presentate le regole di riscrittura per vincoli derivati $dres, rres, dares$ e $rares$. Verranno poi presentate le implementazioni dei vincoli $dres$ e $dares$ dato che le implementazioni, come poi le regole di riscrittura, sono analoghe a quelle dei vincoli $rres$ e $rares$.

***dres*:**

$$dres(A, R, S) \rightarrow \\ un(S, N_1, R) \wedge dom(S, N_2) \wedge N_2 \subseteq A \\ \wedge\, dom(N_1, N_3) \wedge A \parallel N_3 \tag{4.7$_1$}$$

***rres*:**

$$rres(R, A, S) \rightarrow \\ un(S, N_1, R) \wedge ran(S, N_2) \wedge N_2 \subseteq A \\ \wedge\, ran(N_1, N_3) \wedge A \parallel N_3 \tag{4.7$_2$}$$

***dares*:**

$$dares(A, R, S) \rightarrow \\ dres(A, R, T) \wedge un(S, T, R) \wedge S \parallel T \tag{4.7$_3$}$$

***rares*:**

$$rares(R, A, S) \rightarrow rres(R, A, T) \wedge un(S, T, R) \wedge S \parallel T \tag{4.7$_4$}$$

dove N_1, N_2, N_3 e T sono nuovi oggetti logici non inizializzati, A è un insieme logico mentre R ed S sono relazioni binarie.

4.7.1 Implementazioni in JSetL

***dres*:**

```
public Constraint dres(LRel s,LSet a){
 LSet N3 = new LSet();
 LSet N2 = new LSet();
 LRel N1 = new LRel();
 Constraint dom1 = s.dom(N2);
 Constraint dom2 = N1.dom(N3);
 Constraint un = this.union(N1, s);
 Constraint sub = N2.subset(a);
 Constraint disj = a.disj(N3);
 return dom1.and(dom2).and(un).and(sub).and(disj);
}
```

L'implementazione del vincolo *rres* non verrà mostrata, in quanto riprende l'implementazione del vincolo *dres* con la differenza che utilizziamo il vincolo *ran* al posto del vincolo *dom*.

***dares*:**

```
public Constraint dares(LRel s,LSet a){
   LRel T = new LRel();
   Constraint dres = this.dres(s, a);
   Constraint un = this.union(T, s);
   Constraint disj = s.disj(T);
   return dres.and(un).and(disj);
```

```
}
```

L'implementazione del vincolo *rares* non verrà mostrata, in quanto riprende l'implementazione del vincolo *dares* con la differenza che utilizziamo il vincolo *rres* al posto del vincolo *dres*.

4.8 Vincoli negativi

Di seguito verranno presentati i vincoli derivati che rappresentano i "negativi" dei vincoli dom, ran, id, inv e *comp*; ovvero verranno presentate le regole di riscrittura e le implementazioni dei vincoli $ndom, nran, nid, ninv$ e *ncomp*. Anche questi vincoli sono definiti come vincoli derivati.

***ndom*:**

$$ndom(R, A) \rightarrow \bigvee \begin{array}{l} (n_1, n_2) \in R \wedge n_1 \notin A \\ n_1 \in A \wedge comp(\{(n_1, n_1)\}, R, \emptyset) \end{array} \qquad (dom_{12})$$

***nran*:**

$$nran(R, A) \rightarrow \bigvee \begin{array}{l} (n_1, n_2) \in R \wedge n_2 \notin A \\ n_1 \in A \wedge comp(R, \{(n_1, n_1)\}, \emptyset) \end{array} \qquad (ran_7)$$

***nid*:**

$$nid(A, f) \rightarrow \begin{array}{l} n_1 \in A \wedge (n_1, n_1) \notin f \bigvee \\ n_1 \notin A \wedge (n_1, n_1) \in f \bigvee \\ n_1 \neq n_2 \wedge (n_1, n_2) \in f \end{array} \qquad (id_7)$$

***ninv*:**

$$ninv(R, S) \rightarrow \begin{array}{l} (n_1, n_2) \in R \wedge (n_1, n_2) \notin S \bigvee \\ (n_1, n_2) \notin R \wedge (n_1, n_2) \in S \end{array} \qquad (5^{\smile})$$

***ncomp*:**

$$ncomp(R, S, T) \rightarrow \begin{array}{l} (n_1, n_2) \in R \wedge (n_2, n_3) \in S \wedge (n_1, n_3) \notin T \bigvee \\ (n_1, n_3) \in R \wedge comp(\{(n_1, n_1)\}, R, N_1) \\ \wedge comp(S, \{(n_3, n_3)\}, N_2) \\ \wedge comp(N_1, N_2, \emptyset) \end{array} \qquad (\circ_{12})$$

4.8.1 Implementazioni in JSetL

Di seguito verranno mostrate le implementazioni dei vincoli qui sopra mostrati. Dato che sono vincoli derivati anche loro sono stati implementati nella classe `LRel`.

***ndom*:**

```
public Constraint ndom(LSet a){
LVar n1 = new LVar();
LVar n2 = new LVar();
LPair p = new LPair(n1, n2);
LRel m = LRel.empty().ins(new LPair(n1,n1));
LRel empty = LRel.empty();
   Constraint part1 = p.in(this).and(n1.nin(a));
   Constraint part2 = n1.in(a).and(m.comp(this, empty));
   return part1.or(part2);
}
```

***nran*:**

```
 public Constraint nran(LSet a){
   LVar n1 = new LVar();
   LVar n2 = new LVar();
   LPair p = new LPair(n1, n2);
   LRel m = LRel.empty().ins(new LPair(n1,n1));
   LRel empty = LRel.empty();
   Constraint part1 = p.in(this).and(n2.nin(a));
   Constraint part2 = n1.in(a).and(m.comp(this, empty));
   return part1.or(part2);
 }
```

***nid*:**

```
   public Constraint nid(LSet a){
      LVar n1 = new LVar();
      LVar n2 = new LVar();
      LPair p1 = new LPair(n1, n1);
      LPair p2 = new LPair(n2, n1);
      Constraint part1 = n1.in(a).and(p1.nin(this));
```

```
    Constraint part2 = n1.nin(a).and(p1.in(this));
    Constraint part3 = n1.neq(n2).and(p2.in(this));
    return part1.or(part2).or(part3);
  }
```

***ninv*:**

```
public Constraint ninv(LRel s){
 LVar n1 = new LVar();
 LVar n2 = new LVar();
 LPair p1 = new LPair(n1, n2);
 LPair p2 = new LPair(n2, n1);
 Constraint part1 = p1.in(this).and(p2.nin(s));
 Constraint part2 = p1.nin(this).and(p2.in(s));
 return part1.or(part2);
}
```

***ncomp*:**

```
  public Constraint ncomp(LRel s,LRel t){
    LVar n1 = new LVar();
    LVar n2 = new LVar();
    LVar n3 = new LVar();
    LPair p1 = new LPair(n1, n2);
    LPair p2 = new LPair(n2, n3);
    LPair p3 = new LPair(n1, n3);
    LRel m1 = LRel.empty().ins(new LPair(n1, n1));
    LRel m2 = LRel.empty().ins(new LPair(n3, n3));
    LRel N1 = new LRel();
    LRel N2 = new LRel();
    LRel empty = LRel.empty();
    Constraint part1 = p1.in(this).
              and(p2.in(s)).and(p3.nin(t));
    Constraint part2 = p3.in(t).and(m1.comp(this, N1)).
              and(s.comp(m2, N2)).and(N1.comp(N2, empty));
    return part1.or(part2);
  }
```

4.8.2 Esempi d'uso dei vincoli

Mostriamo ora un paio di esempi d'utilizzo per ogni vincolo derivato sopra descritto.

***ndom:* Esempio 1**

```
// creo m1 = {(1,2),(3,4)} di nome "m1"
LRel m1 = LRel.empty().ins(new LPair(1,2)).
          ins(new LPair(3,4)).setName("m1");
// creo a1 LSet non inizializzato di nome "a1"
LSet a1 = new LSet("a1");
solver.add(m1.ndom(a1));
solver.solve();
solver.showStore();
solver.nextSolution(); //stampo un'altra possibuile soluzione
solver.showStore();   // se questa esiste
solver.nextSolution();
solver.showStore();
a1.output();
m1.output();

/*OUTPUT
   Store: 3 nin _a1              // ci sono 3 possibili soluzioni
   Store: 1 nin _a1              // per questo caso del vincolo
   Store: _N8 neq 3 AND _N8 neq 1
   a1 = {_N8|_N22}
   m1 = {[3,4],[1,2]}
 */
```

***ndom:* Esempio 2**

```
// creo m2 LRel non inizializzato di nome "m2"
LRel m2 = new LRel("m2");
// creo a2 = {1,2} di nome "a2"
LSet a2 = LSet.empty().ins(1).ins(2).setName("a2");
solver.add(m2.ndom(a2));
solver.solve();
```

```
solver.showStore();
a2.output();
m2.output();

/*OUTPUT
   Store: _N42 neq 2 AND _N42 neq 1
   a2 = {2,1}
   m2 = {[_N42,_N43]|_N51}
 */
```

nran: Esempio 1

```
// creo m3 = {(1,2),(3,4)} di nome "m3"
LRel m3 = LRel.empty().ins(new LPair(1,2)).
           ins(new LPair(3,4)).setName("m3");
// creo a3 LRel non inizializzato di nome "a3"
LSet a3 = new LSet("a3");
solver.add(m3.nran(a3));
solver.solve();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
a3.output();
m3.output();

/*OUTPUT
   Store: 4 nin _a1
   Store: 2 nin _a1
   Store: _N8 neq 3 AND _N8 neq 1
   a3 = {_N8|_N22}
   m3 = {[3,4],[1,2]}
 */
```

nran: Esempio 2

```
// creo m4 LRel non inizializzato di nome "m4"
LRel m4 = new LRel("m4");
// creo a4 = {1,2} di nome "a4"
LSet a4 = LSet.empty().ins(1).ins(2).setName("a4");
solver.add(m4.nran(a4));
solver.solve();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
a4.output();
m4.output();

/*OUTPUT
   Store: _N43 neq 2 AND _N43 neq 1
   Store: comp({[2,2]},_m4,{})
   Store: comp({[1,1]},_m4,{})
   a4 = {2,1}
   m4 = unknown
 */
```

nid: Esempio 1

```
// creo s1 = {1,2,3} di nome "s1"
LSet s1 = LSet.empty().ins(1).ins(2).ins(3).setName("s1");
// creo r1 LRel non inizializzato di nome "r2"
LRel r1 = new LRel("r1");
solver.add(r1.nid(s1));
solver.solve();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
```

```
solver.nextSolution();
solver.showStore();
s1.output();
r1.output();

/*OUTPUT
   Store: [3,3] nin _r1
   Store: [2,2] nin _r1
   Store: [1,1] nin _r1
   Store: _N5 neq 3 AND _N5 neq 2 AND _N5 neq 1
   Store: _N5 neq _N6
   s1 = {3,2,1}
   r1 = {[_N6,_N5]|_N14}
 */
```

nid: Esempio 2

```
// creo s2 = {4,2,3| S} di nome "s2"
LSet s2 = new LSet().ins(4).ins(2).ins(3).setName("s2");
// creo r2 = {(1,1),(2,2),(3,3)| R} di nome "r2"
LRel r2 = new LRel().ins(new LPair(1,1)).
      ins(new LPair(2,2)).ins(new LPair(3,3)).setName("r2");
solver.add(r2.nid(s2));
solver.solve();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
r2.output();
s2.output();

/*OUTPUT
   Store: [4,4] nin _N84
   Store: [_N94,_N94] nin _N84
   Store: 1 nin _N80
   r2 = {[3,3],[2,2],[1,1]|_N84}
   s2 = {3,2,4|_N80}
```

```
 */
```

ninv: Esempio 1

```
// creo s3 LRel non inizializzato di nome "s3"
LRel s3 = new LRel("s3");
// creo r3 ={(1,2),(2,3),(3,4)} di nome "r3"
LRel r3 = LRel.empty().ins(new LPair(1,2)).
      ins(new LPair(2,3)).ins(new LPair(3,4)).setName("r3");
solver.add(r3.ninv(s3));
solver.solve();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
r3.output();
s3.output();

/*OUTPUT
   Store: [4,3] nin _s3
   Store: [3,2] nin _s3
   Store: [2,1] nin _s3
   Store: []
   r3 = {[3,4],[2,3],[1,2]}
   s3 = {[_N12,_N11]|_N24}
 */
```

ninv: Esempio 2

```
// creo s4 = {(2,1),(3,2),(4,3)} di nome "s4"
LRel s4 = LRel.empty().ins(new LPair(2,1)).
      ins(new LPair(3,2)).ins(new LPair(4,3)).setName("s4");
// creo r4 = {(1,2),(2,3),(3,4)} di nome "r4"
LRel r4 = LRel.empty().ins(new LPair(1,2)).
      ins(new LPair(2,3)).ins(new LPair(3,4)).setName("r4");
```

```
solver.add(r4.ninv(s4));
solver.solve();
solver.showStore();
r4.output();
s4.output();

/*OUTPUT  JSetL.Failure */
```

ncomp: Esempio 1

```
// creo r4 = {(1,2),(3,4)| R} di nome "r5"
LRel r5 = new LRel().ins(new LPair(1,2)).
          ins(new LPair(3,4)).setName("r5");
// creo s5 = {(2,6),(4,8)| S} di nome "s5"
LRel s5 = new LRel().ins(new LPair(2,6)).
          ins(new LPair(4,8)).setName("s5");
// creo q5 = {(5,7)| Q} di nome "q5"
LRel q5 = new LRel().ins(new LPair(5,7)).setName("q5");
solver.add(r5.ncomp(s5, q5));
solver.solve();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
solver.nextSolution();
solver.showStore();
r5.output();
s5.output();
q5.output();

/*OUTPUT
  Store: [3,8] nin _N102
  Store: [3,_N108] nin _N102
```

```
  Store: [1,6] nin _N102
  Store: [1,_N108] nin _N102
  Store: [_N106,8] nin _N102
  Store: [_N106,6] nin _N102
  r5 = {[3,4],[1,2],[_N106,2]|_N142}
  s5 = {[4,8],[2,6]|_N95}
  q5 = {[5,7]|_N102}
*/
```

ncomp: Esempio 2

```
// creo r6 = {(1,2),(3,4)} di nome "r6"
LRel r6 = LRel.empty().ins(new LPair(1,2)).
          ins(new LPair(3,4)).setName("r6");
// creo s6 = {(1,2),(2,3),(3,4)} di nome "s6"
LRel s6 = LRel.empty().ins(new LPair(5,6)).
          ins(new LPair(7,8)).setName("s6");
// creo q6 = {} di nome "q6"
LRel q6 = LRel.empty().setName("q6");
solver.add(r6.ncomp(s5, q6));
solver.solve();
r6.output();
s6.output();
q6.output();

/*OUTPUT  JSetL.Failure */
```

5 Riscrittura dei vincoli su funzioni parziali

In questo capitolo verranno presentate le regole di riscrittura e le rispettive implementazioni dei vincoli su `LMap`. La sintassi utilizzata per le regole di riscrittura dei vincoli corrisponde a quella vista nel capitolo precedente. Il vincolo *ran* per `LMap` sfrutta la stessa implementazione e regole di quello mostrato per le `LRel` per tanto qui non viene riproposto.

5.1 Il vincolo *dom* per `LMap`

Il vincolo di dominio nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ per le funzioni parziali ha la stessa forma ($dom(f, a)$) e semantica di quello già visto per le relazioni binarie. Le regole di riscrittura variano rispetto a quelle viste per lo stesso vincolo visto in `LRel` e sono le seguenti:

$$dom(\dot{f}, \dot{f}) \rightarrow \dot{f} = \emptyset \qquad (dom_7)$$

$$dom(f, \emptyset) \rightarrow f = \emptyset \qquad (dom_8)$$

$$dom(\emptyset, A) \rightarrow A = \emptyset \qquad (dom_9)$$

$$dom(\dot{f}, \{x \sqcup A\}) \rightarrow \dot{f} = \{(x, n) \sqcup N\} \wedge dom(N, A) \qquad (dom_{10})$$

$$dom(\{(x, y) \sqcup f\}, A) \rightarrow A = \{x \sqcup N\} \wedge dom(R, N) \qquad (dom_{11})$$

dove N è un nuovo oggetto logico non inizializzato, A è un insieme logico ed f una funzione parziale. Le regole fornite sfruttano il principio di ricorsione; le prime tre rappresentano i casi base mentre le regole $10, 11$ rappresentano i casi riscorsivi.
$dom(\dot{f}, \dot{A})$, dove $\dot{f}$ e $\dot{A}$ sono variabili, è l'unica forma irriducibile del vincolo *dom* per `LMap`.

5.1.1 Implementazione in JSetL

```
private void mapDomainRule(LSet a,LSet r, AConstraint s)
     throws Failure{
  manageEquChains(s);

    /**CASI BASE*/

    //Implementazione delle regole 7 e 8
    // dom(r,r) or dom(r,{})
    if(a.equals(r) || (a.isInit() && a.isEmpty())){
        s.arg1 = r;
        s.arg2 = LMap.empty();
        s.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }
    //Implementazione della regola9
    //dom({},a)
    if(r.isInit() && r.isEmpty()){
        s.arg2 = a;
        s.arg1 = LMap.empty();
        s.cons = Environment.eqCode;
        Solver.storeUnchanged = false;
        return;
    }

    /**CASI RICORSIVI*/

    //Implementazione della regola 10
    //dom(r,a) con r variabile e a != {}
    if(r.isInit()){
        LPair tmp = (LPair)r.getOne();
        LSet rest = new LSet();
        Solver.add(r.eq(rest.ins(tmp)));
        Solver.add(rest.ncontains(tmp));
        LSet rs = new LSet();
        Object t = tmp.getFirst();
```

```
            LSet tmpDom;
            if(t != null){
                tmpDom = rs.ins(t);
            }else{
               LVar x = new LVar();
                tmpDom = rs.ins(x);
                Solver.add(tmp.eq(new LPair(x, new LVar())));
            }
            Solver.add(new AConstraint(a,Environment.eqCode,tmpDom));
            s.arg1 = rest;
            s.arg2 = rs;
            Solver.storeUnchanged = false;
            return;
        }
        //Implementazione della regola 11
        //dom(r,a) dove r != {} e a non  specificato
        if(a.isInit()){
         Object tmp = a.getOne();
            LSet rest = new LSet();
            LMap rs = new LMap();
            Solver.add(a.eq(rest.ins(tmp)));
            LMap tmpMap = rs.ins(new LPair(tmp,new LVar()));
            Solver.add(new AConstraint(r,Environment.eqCode,tmpMap));
            Solver.add(new AConstraint(tmp,Environment.ninCode,rest));
            s.arg1 = rs;
            s.arg2 = rest;
            Solver.storeUnchanged = false;
            return;
        }
    }
```

5.1.2 Esempi d'uso del vincolo

Dato che il funzionamento di *dom* per `LMap` è simile a quello per `LRel`, di seguito viene mostrato solo un caso in cui l'output del *solver* differisce dagli esempi visti per lo stesso vincolo in `LRel`.

Esempio 1

```
// creo m1 = {(1,2),(3,4)| R} di nome "m1"
LMap m1 = new LMap("R").ins(new LPair(1,2)).
          ins(new LPair(3,4)).setName("m1");
// creo d1 = {1,5| D} con nome "d1"
LSet d1 = new LSet("D").ins(1).ins(5).setName("d1");
solver.add(m1.dom(d1));
solver.solve();
solver.showStore();
solver.showStore();
solver.showStore();

m1.output();
d1.output();

/* OUTPUT
   Store: mapDom(_N121,_N75) AND 5 nin _N75
   m1 = {[3,4],[1,2],[5,_N122]|_N121}
   d1 = {5,1,3|_N75}
*/
```

Come si può notare, a differenza di quanto visto per il vincolo *dom* in `LRel`, qui viene applicata la definizione di funzione parziale e di conseguenza viene rischiesto non solo che il resto di `d1` sia il dominio del resto di `r1`, ma anche che il valore 5 non si ripresenti all'interno del resto del dominio.

5.2 Il vincolo *comp* per `LMap`

Il vincolo di composizione sequenziale nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ per le funzioni parziali ha la stessa forma $comp(f, g, h)$ e semantica $(h = f \circ g)$ di quello già visto per le relazioni binarie. Le regole di riscrittura variano rispetto a quelle viste per lo stesso vincolo visto in `LRel` e sono le seguenti:

$$comp(\emptyset, g, h) \rightarrow h = \emptyset \qquad (\circ_7)$$

$$comp(f, \emptyset, h) \rightarrow h = \emptyset \qquad (\circ_8)$$

$$comp(f, g, \emptyset) \rightarrow ran(f, N_1) \wedge dom(g, N_2) \wedge N_1 \| N_2 \qquad (\circ_9)$$

$$\begin{aligned} &comp(\{(x, y) \sqcup f\}, g, \dot{h}) \rightarrow \\ &\quad g = \{(y, n) \sqcup N_1\} \wedge \dot{h} = \{(x, n) \sqcup N_2\} \wedge comp(f, g, N_2) \bigvee \\ &\quad dom(g, N_1) \wedge y \notin N_1 \wedge comp(f, g, \dot{h}) \end{aligned} \qquad (\circ_{10})$$

$$\begin{aligned} &comp(f,g, \{(x, y) \sqcup h\}) \rightarrow \\ &\quad f = \{(x, n) \sqcup N_1\} \wedge g = \{(n, z) \sqcup N_2\} \wedge comp(N_1, g, h) \end{aligned} \qquad (\circ_{11})$$

dove N_1 e N_2 sono nuovi oggetti logici non inizializzati, f, g e h sono funzioni parziali. Le regole fornite sfruttano il principio di ricorsione; le prime due rappresentano i casi base mentre le regole $9, 10, 11$ rappresentano i casi riscorsivi.

Forme irriducibili del vincolo:

- $comp(\dot{f}, g, \dot{h})$ con g $\neq \emptyset$
- $comp(f, \dot{g}, \dot{h})$ con f $\neq \emptyset$

dove $\dot{f}, \dot{g}, \dot{h}$ sono variabili.

Notare che le regole, qui sopra presentate, sono più semplici rispetto a quelle mostrate per le relazioni binarie, in particolare la regola 11 è molto più semplice della corrispondente regola 6.

5.2.1 Implementazione in JSetL

```
private void map_compRule(LSet r,LSet s, LSet q, AConstraint c)
     throws Failure{
    manageEquChains(c);

    /**CASI BASE*/

    //Implementazione della regola 1
    //comp({},s,q)
    if(r.isInit() && r.isEmpty()){
        Solver.add(new AConstraint(q,Environment.eqCode,LMap.empty()));
        c.solved=true;
        return;
    }
    //Implementazione della regola 2
    //comp(r,{},q)
    if(s.isInit() && s.isEmpty()){
        Solver.add(new AConstraint(q,Environment.eqCode,LMap.empty()));
        c.solved=true;
        return;
    }

    /**CASI RICORSIVI*/

    //Implementazione della regola 3
    //comp(r,s,{})
    if(q.isInit() && q.isEmpty() && s.isInit() && !s.isEmpty()
        && r.isInit() && !r.isEmpty()){
     LSet rr = new LSet();
        LSet ds = new LSet();
        Solver.add(new AConstraint(r,Environment.pfRanCode,rr));
        Solver.add(new AConstraint(s,Environment.pfDomCode,ds));
        Solver.add(new AConstraint(rr,Environment.disjCode,ds));
        c.solved = true;
        return;

    }
```

```
//Implementazione della regola 5
//comp(r,s,q) dove q != {} mentre s,r non sono specificate
if(q.isInit() && !q.isEmpty()) {
    LPair hq = (LPair) q.getOne();
    LMap rq = (LMap)q.removeOne();
    LMap N1= new LMap();
    LMap N2 = new LMap();
    LVar n = new LVar();
    LPair newRLPair = new LPair(hq.getFirst(),n);
    LMap appR = N1.ins(newRLPair);
    LPair newSLPair = new LPair(n,hq.getSecond());
    LMap appS = N2.ins(newSLPair);
    Solver.add(new AConstraint(r,Environment.eqCode,appR));
    Solver.add(new AConstraint(s,Environment.eqCode,appS));
    c.arg1 = N1;
    c.arg3 = rq;
    map_comp(c);
    Solver.storeUnchanged = false;
    return;
}
//Implementazione della regola 4
////comp(r,s,q) dove q  una variabile, s non  specificata e r != {}
if(r.isInit() && !r.isEmpty() && !q.isInit()){
    LPair hr = (LPair)r.getOne();
    LSet rr = r.removeOne();
    switch (c.caseControl) {
    case 0:
        VarState statoVarPfun = Solver.B.getVarState();
        StoreState statoStorePfun = Solver.B.getStoreState(c);
        Solver.B.addChoicePoint(1, statoVarPfun, statoStorePfun);
        LVar n = new LVar();
        LPair yn = new LPair(hr.getSecond(),n);
        LPair xn = new LPair(hr.getFirst(),n);
        LMap N1 = new LMap();
        LMap N2 = new LMap();
        Solver.add(new AConstraint(s,Environment.eqCode,N1.ins(yn)));
        Solver.add(new AConstraint(q,Environment.eqCode,N2.ins(xn)));
        c.arg1 = rr;
```

```
            c.arg3 = N2;
            map_comp(c);
            Solver.storeUnchanged = false;
            return;
        case 1:
            c.caseControl = 0;
            LSet doms = new LSet();
            LMap N1_ = new LMap();
            Solver.add(new AConstraint(s,Environment.pfDomCode,doms));
            Solver.add(new AConstraint(hr.getSecond(), Environment.ninCode,
                N1_));
            c.arg1 = rr;
            map_comp(c);
            Solver.storeUnchanged = false;
            return;
        }
    }
}
```

5.2.2 Esempi d'uso del vincolo

Esempio 1

```
// creo r2 LMap non inizializzato con nome "r2"
LMap r2 = new LMap("r2");
// creo s2 LMap non inizializzato con nome "s2"
LMap s2 = new LMap("s2");
// creo q2 = {(1,6),(3,8)| R} di nome "q2"
LMap q2 = new LMap("R").ins(new LPair(1,6)).
          ins(new LPair(3,8)).setName("q2");
solver.add(r2.comp(s2, q2));
solver.solve();
solver.showStore();
r2.output();
s2.output();
q2.output();
```

```
/*OUTPUT
   Store: comp(_N197,{[_N175,8],[_N199,6]|_N248},_N139)
   r2 = {[3,_N175],[1,_N199]|_N197}
   s2 = {[_N175,8],[_N199,6]|_N248}
   q2 = {[3,8],[1,6]|_N139}
 */
```

Esempio 2

```
// creo r3 = {(1,2),(3,4)| R} di nome "r3"
LMap r3 = new LMap("R1").ins(new LPair(1,2)).
          ins(new LPair(3,4)).setName("r3");
// creo s3 = {(2,6),(4,8)| R} di nome "s3"
LMap s3 = new LMap("R2").ins(new LPair(2,6)).
          ins(new LPair(4,8)).setName("s3");
// creo q3 LRel non inizializzato con nome "q3"
LMap q3 = new LMap("q3");
solver.add(r3.comp(s3, q3));
solver.solve();
solver.showStore();
r3.output();
s3.output();
q3.output();

/*OUTPUT
   Store: _N1986 = _R1 comp {[4,8],[2,6]|_R2}
   r3 = {[3,4],[1,2]|_R1}
   s3 = {[4,8],[2,6]|_R2}
   q3 = {[3,8],[1,6]|_N1986}
 */
```

5.3 Il vincolo *pfun* per LMap

Il vincolo di funzione parziale nel linguaggio $\mathcal{L}_{\mathcal{BR}}$ ha la forma $pfun(f)$ dove f è una funzione parziale. Il vincolo è soddisfatto se e solo se f soddisfa le condizioni di funzione parziale ovvero:

$$\forall x, y, y' : ([x, y] \in f \wedge [x, y'] \in f \implies y = y')$$

Le regole di riscrittura per il vincolo $pfun$ sono le seguenti:

$$pfun(\emptyset) \rightarrow true \qquad (\mapsto_1)$$

$$\left(\begin{array}{l} if\ \ f = \{t_1 \cdots t_n\}\ \ AND\ \ n \geq 1\ \ AND \\ is\_ground(dom(f))\ \ AND\ \ is\_pfun(f) \end{array} \right) pfun(f) \rightarrow true \qquad (\mapsto_2)$$

$$pfun(\{(x,y) \sqcup f\}) \rightarrow \{(x,y) \sqcup f\} = \{(x,y) \sqcup F\} \wedge (x,y) \notin F \wedge dom(F, d_F) \wedge x \notin d_F \wedge pfun(F) \qquad (\mapsto_3)$$

dove F e d_F sono nuovi oggetti logici non inizializzati, f funzione parziale. Le regole fornite sfruttano il principio di ricorsione; le prime due rappresentano i casi base mentre la regola 3 rappresenta il caso riscorsivo.
$pfun(\dot{f})$, dove $\dot{f}$ è una variabile, è l'unica forma irriducibile del vincolo $pfun$.

5.3.1 Implementazione in JSetL

```
private void pfunRule(LSet r,AConstraint s) throws Failure{
    manageEquChains(s);

    /**CASI BASE*/

    //implementazione della regola 1
    // r = {}
    if(r.isInit() && r.isEmpty()){
        s.solved = true;
        return;
    }
    //implementazione della regola 2
    // r = {(x1,y1),...,(xn,yn)}
    if(r.isInit() && LMap.isDomainGround(r))
     if(LMap.isPfun(r)){
        s.solved = true;
        return;
     }
     else
        Solver.fail(s);
    //implementazione della regola 3
    // r = {(x,y)| R}
    if(r.isInit()) {
     LPair head = (LPair) r.getOne();
     LMap R1 = new LMap();
     LSet D = new LSet();
     AConstraint un = new AConstraint(r, Environment.eqCode, R1.ins(head));
     Solver.add(un);
     Solver.add(R1.dom(D));
     Solver.add(head.nin(R1));
     Solver.add(D.ncontains(head.getFirst()));
     s.arg1=R1;
     Solver.storeUnchanged = false;
     return;
    }
}
```

5.3.2 Esempi d'uso del vincolo

Esempio 1

```
LVar x = new LVar("X");
//creo m1 = {(1,5),(X,3)}
LMap m1 = LMap.empty().ins(new LPair(1,5)).ins(new LPair(x,3)).setName("m1
    ");
solver.add(x.eq(2));
solver.add(m1.pfun());
solver.solve();
solver.showStore();
m1.output();

/*OUTPUT
   Store: []
   m1 = {[2,3],[1,5]}
 */
```

Esempio 2

In questo esempio assumiamo che vi siano le stesse dichiarazioni dell'esempio 1 ma, invece di invocare il vincolo `x.eq(2)`, in questo caso, viene invocato il vincolo `x.eq(1)`.

```
solver.add(x.eq(1));
solver.add(m1.pfun());
solver.solve();
solver.showStore();
m1.output();

/*OUTPUT JSetL.Failure */
```

Esempio 3

```
//creo m3 LMap con nome "m3"
LMap m3 = new LMap("m3");
solver.add(m3.pfun());
solver.solve();
solver.showStore();
m3.output();

/*OUTPUT
     Store: pfun(_m3)   \forma risolta
  m3 = unknown
 */
```

5.4 Dettagli implementativi sui vincoli presentati

Verranno presentati di seguito alcuni particolari sull'implementazione di alcuni vincoli su `LRel` e su `LMap` precedentemente descritti.

5.4.1 Vincolo *ran*

L'implementazione del vincolo *ran* viene condivisa sia da `LRel` che da `LMap`. Dato che queste due classi necessitano di usare due implementazioni diverse del vincolo *comp*, è necessario che venga richiamato il metodo corretto in base al tipo degli oggetti che si stà utilizzando.
Purtroppo, in casi particolari, non è possibile distinguere tra i due tipi di dato, e quindi, si è deciso di avere, a livello di implementazione, due vincoli con due id diversi, uno per `LRel` e uno per `LMap`.
Nell'implementazione della regola ran_5 del vincolo, vengono distinti i due casi possibili in base all'id del vincolo di *ran* creando, in ogni caso, gli oggetti di tipo opportuno e richiamando il vincolo *comp* corretto. Dal punto di vista del risolutore tutto ciò vien visto come se esistesssero due vincoli *ran*, ma effettivamente si ha un'unica implementazione in grado di distinguere in modo corretto i due casi.

5.4.2 Parametri di tipo `LSet`

Quando viene richiamato un vincolo di `LSet`, durante il processo di risoluzione di un vincolo su `LRel` o su `LMap`, si possono perdere informazioni riguardanti il tipo di dato degli oggetti di partenza, in quanto il campo `equ` (su cui effettivamente si opera) di tali oggetti potrebbe essere di tipo `LSet`.
Per questo motivo le dichiarazioni dei metodi che implementano i vincoli primitivi su `LRel` e `LMap` prevedono parametri del tipo piu' generale `LSet`. Per non confondere le implementazioni dei vincoli di `LRel` da quelle di `LMap` si dovrà pertanto fare affidamento sull'id del vincolo richiamato (vedi ad esempio vincolo *ran*).

6 Conclusioni e sviluppi futuri

In questa tesi è stato mostrato come integrare il linguaggio logico $\mathcal{L}_{\mathcal{BR}}$ con vincoli su relazioni binarie e funzioni parziali, all'interno di un linguaggio di programmazione ad oggetti come Java attraverso l'utilizzo della libreria JSetL. La presenza in JSetL dell'implementazione del linguaggio CLP($\mathcal{SET}$), che fornisce insiemi logici con i vincoli ad essi associati, ha permesso di integrare relazioni binarie e funzioni parziali in modo relativamente semplice all'interno della libreria sfruttandone le capacità di rappresentazione e manipolazione di insiemi e aggiungendo i vincoli e le classi necessarie per trattare le nuove strutture dati.

Per quanto riguarda i lavori futuri, un aspetto sicuramente da migliorare riguarda l'implementazione delle regole di riscrittura usate per il vincolo *comp* su `LRel`, che risultano estremamente inefficienti; pertanto si stà provvedendo a individuare un insieme di regole più efficienti per tale vincolo.
Un altro possibile sviluppo è quello di migliorare l'integrazione tra le relazioni binarie e altre astrazioni di recente aggiunte a JSetL (come i cosiddetti Restricted Intensional Sets) o ancora in corso di realizzazione (come il Prodotto Cartesiano tra insiemi). Per quest'ultimo, in particolare, si può pensare di definirlo come estensione della classe `LRel` e quindi applicare ad esso tutte le operazioni e i vincoli previsti in JSetL per le relazioni binarie.

Riferimenti bibliografici

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

[2] JSetL Home Page
http://cmt.math.unipr.it/jsetl.html

[3] Maximiliano Cristià e Gianfranco Rossi
Rewrite Rules for a Solver for Set, Binary Relations and Partial Functions
June 22, 2018
http://people.dmi.unipr.it/gianfranco.rossi/SETLOG/calculus.pdf

[4] Maximiliano Cristià e Gianfranco Rossi
Solving quantifier-free first-order constraints over finite sets and binary relations (Manoscritto, in corso di pubblicazione)

[5] Gianluca Lutero
Estensione della libreria Java JSetL con vincoli su funzioni parziali
Tesi di laurea del Corso di Laurea in Informatica, Università di Parma, marzo 2016