UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI
MATEMATICA E INFORMATICA

Corso di Laurea in Informatica

Tesi di Laurea

# Estensione della libreria Java JSetL con vincoli su funzioni parziali

Relatore:
**Prof. Gianfranco Rossi**

Candidato:
**Gianluca Lutero**

Anno Accademico 2014/2015

*Ai miei genitori,*
*Domenica e Giuseppe*
*e a mio fratello*
*Giacomo*

# Indice

# Introduzione

Dati due insiemi $X$ e $Y$, una *relazione binaria* tra $X$ e $Y$ è un qualsiasi sottoinsieme di $\wp(X \times Y)$. Le funzioni parziali sono un caso particolare di relazione binaria, in cui coppie ordinate devono verificare la nozione classica di funzione, ovvero ogni elemento del dominio deve essere mappato in al più un elemento del codominio oppure indefinito. Le relazioni binarie invece sono solo insiemi di coppie ordinate. Tutti gli operatori relazionali (come dom e ran) e tutti gli operatori insiemistici possono essere applicati su entrambe. Le funzioni si distinguono dalle relazioni binarie in quanto se $x$ è un elemento del dominio di una funzione parziale $f$ allora $f(x)$ è definito come quell'elemento $y \in range(f)$ tale che $(x, y) \in f$.

Le funzioni parziali sono una nozione fondamentale nella matematica, ma rappresentano anche una conveniente astrazione nell'ambito della programmazione e nella specifica dei programmi. Nei linguaggi di specifica formale come Z, B e Alloy la nozione di funzione è utilizzata principalmente per modellare sistemi state-based. Le specifiche vengono viste come formule che coinvolgono operazioni su insiemi e funzioni parziali.

Nei linguaggi di programmazione eseguibili non viene dato grande supporto per le funzioni parziali rappresentate nella notazione estensionale, un esempio sono alcune funzionalità offerte dalle librerie Java e C++ che forniscono strutture dati associative come la classe `Map`.

Questi linguaggi, tuttavia, trattano strutture dati *completamente specificate* dove ogni elemento della funzione è definito e le operazioni che si applicano sono solo quelle su funzioni completamente specificate.
Come accade per i tipi di dato più convenzionali molte applicazioni possono beneficiare dell'utilizzo delle funzioni parziali in questo modo generale dove esse o parti di esse possono non essere specificate. Questo può essere ottenuto spostando il contesto dalla programmazione tradizionale alla programmazione per vincoli, dove le funzioni parziali possono essere rappresentate da variabili *unbound* e le operazioni sulle stesse sono espresse come vincoli.

La libreria JSetL[1, 7] introduce queste funzionalità di programmazione per vincoli nel linguaggio Java. Lo scopo di questa tesi è mostrare un'imple-

mentazione delle funzioni parziali e dei vincoli ad esse associati introducendole come struttura dati primitive in JSetL. La rappresentazione delle funzioni parziali in JSetL rispecchia la definizione data in precedenza di insiemi di coppie grazie alla presenza degli insiemi logici e dei vincoli sugli insiemi già presenti nel linguaggio.

# Capitolo 1

# Funzioni parziali

## 1.1 Un linguaggio a vincoli su insiemi e funzioni parziali

Il linguaggio a vincoli CLP($\mathcal{PF}$) definisce la struttura e i vincoli sulle funzioni parziali. Il linguaggio segue la definizione data di funzione parziale come estensione degli insiemi, per cui CLP($\mathcal{PF}$) estende il linguaggio a vincoli sugli insiemi denominato CLP($\mathcal{SET}$) [2, 4, 8].

### 1.1.1 Il linguaggio CLP($\mathcal{SET}$)

**Sintassi**

La sintassi del linguaggio CLP($\mathcal{SET}$) è data dalla quadrupla $\Sigma_{\mathcal{SET}} = \langle \mathcal{F}, \Pi_C, \Pi_U, \mathcal{V} \rangle$. Ogni elemento è definito come segue:

- $\mathcal{F}$ è un insieme di costanti e simboli di funzione definito come segue:
  - $\emptyset \in \mathcal{F}$
  - $\{\cdot|\cdot\} \in \mathcal{F}$ e $int \in \mathcal{F}$, simboli di relazione binaria
  - $\mathcal{F}' \subset \mathcal{F}$ dove $\mathcal{F}'$ è un insieme di simboli di funzione
- $\Pi_C = \{=, in, un, disj, \leq, size, set, integer\}$, insieme di simboli di predicati
- $\Pi_U$ è un insieme di simboli di predicati definito dall'utente
- $\mathcal{V}$ è un insieme numerabile di simboli di variabile

**Semantica informale**

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

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

$$int(m,n) = \begin{cases} \emptyset & \text{se } m > n \\ [m,n] & \text{se } m \leq n \end{cases} \tag{1.1}$$

- $\mathcal{F}' = \{Z, F_Z, F_U\}$ rappresenta l'unione degli insiemi che rappresentano rispettivamente i numeri interi, le operazioni sui numeri interi, le costanti definite dall'utente e i simboli di funzione con la seguente proprietà $F_U \cap (Z \cup F_Z \cup \{\emptyset, \{\cdot|\cdot\}, int\}) = \emptyset$
- 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 segue: $un(r,s,t) = true \iff t = r \cup s$
- Il predicato *disj* rappresenta la relazione di disgiunzione insiemistica definita come $disj(r,s) = true \iff r \cap s = \emptyset$
- Il predicato *size* rappresenta la relazione cardinalità di insiemi definita come $size(s,n) = true \iff n = |s|$
- Il predicato $\leq$ rappresenta la relazione di confronto "minore o uguale" sugli interi
- Il predicato *set* controlla che il termine sia un insieme.

**Vincoli**

Una qualsiasi combinazione dei letterali $\langle \mathcal{F}, \Pi_C, \mathcal{V} \rangle$ è un $\mathcal{SET}$*-constraint* primitivo. Una congiunzione di vincoli primitivi è detta $\mathcal{SET}$*-constraint* composto.
Le seguenti formule sono un esempio di $\mathcal{SET}$*-constraint*:

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

### 1.1.2 Il linguaggio CLP($\mathcal{PF}$)

Il linguaggio a vincoli sulle funzioni parziali estende il linguaggio CLP($\mathcal{SET}$) precedentemente descritto. Il nuovo dominio dei vincoli e il relativo linguaggio vengono chiamati rispettivamente $\mathcal{PF}$ e CLP($\mathcal{PF}$) . Siccome $\mathcal{PF}$ include $\mathcal{SET}$ come caso particolare verrà mostrato solo ciò che viene aggiunto in $\mathcal{PF}$ rispetto a $\mathcal{SET}$. Le funzioni parziali sono un particolare tipo di insiemi, per cui non viene introdotto alcun nuovo simbolo ma vengono usati opportuni vincoli sugli elementi degli insieme che le rappresentano.

#### Funzioni parziali

*Un insieme r rappresenta una funzione parziale se r assume una delle seguenti forme:*

- $r = \{\}$
- $r = \{[x_1, t_1], [x_2, t_2], \ldots, [x_n, t_n]\}$
- $r = \{[x_1, t_1], [x_2, t_2], \ldots, [x_n, t_n] | s\}$

*e* $x_i, t_i, i = 1, \ldots, n$ *sono termini e vale il vincolo*
$\forall i, j = 1, \ldots, n \wedge i \neq j : x_i \neq x_j, x_i \notin dom(s)$

#### Vincoli primitivi

Si è scelto di aggiungere all'insieme $\Pi_C$ solo alcuni simboli di predicato che trattano le funzioni parziali. Il nuovo insieme di vincoli primitivi diventa quindi $\Pi_C = \{=, in, un, disj, \leq, size, set, integer\} \cup \{dom, ran, pfun, comp\}$. Di seguito è data una semantica informale dei nuovi vincoli introdotti.

- *dom(r,A)*: dove $r$ è una funzione parziale e $A$ un insieme che rappresenta il suo dominio. L'interpretazione intuitiva del vincolo è

$$dom(r, A) = true \longleftrightarrow r = \{[x, y] | rr\} \wedge x \in A$$

- *ran(r,A)*: dove $r$ è una funzione parziale e $A$ un insieme che rappresenta il suo codominio. L'interpretazione intuitiva del vincolo è

$$ran(r, B) = true \longleftrightarrow r = \{[x, y] | rr\} \wedge y \in B$$

- *pfun(r)*: dove $r$ è una funzione parziale. L'interpretazione intuitiva del vincolo è

$$pfun(r) = true \longleftrightarrow \forall x, y, y' : ([x, y] \in r \wedge [x, y'] \in r \rightarrow y = y')$$

- *comp(r,s,t)*: dove $r, s$ e $t$ sono funzioni parziali. L'interpretazione intuitiva del vincolo è

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

I $(\mathcal{SET}, \mathcal{PF})$ – *constraint* primitivi sono sufficienti per definire molte delle comuni operazioni su di esse come congiunzione di vincoli primitivi[5]. Precisamente valgono le seguenti equivalenze:

- ndres(a,r,s) $\Leftrightarrow$ dres(a,r,b) $\wedge$ diff(r,b,s)
- nrres(b,r,s) $\Leftrightarrow$ rres(b,r,a) $\wedge$ diff(r,a,s)
- dres(a,r,s) $\Leftrightarrow$ dom(r,dr) $\wedge$ dom(s,ds) $\wedge$ inters(a,dr,ds) $\wedge$ subset(s,r)
- rres(b,r,s) $\Leftrightarrow$ un(s,t,r) $\wedge$ ran(s,rs) $\wedge$ ran(r,rr)
  $\wedge$ inters(b,rr,rs) $\wedge$ ran(t,rt) $\wedge$ disj(rs,rt)
- rimg(b,r,s) $\Leftrightarrow$ dres(b,r,rb) $\wedge$ ran(rb,s)
- oplus(r,s,t) $\Leftrightarrow$ un(rs,s,t) $\wedge$ ndres(ds,r,rs) $\wedge$ dom(s,ds)
- id(a,r) $\Leftrightarrow$ dom(r,a) $\wedge$ ran(r,a) $\wedge$ comp(r,r,r)

### 1.1.3 Il solver $SAT_{\mathcal{PF}}$

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

---

**procedure** $SAT_{\mathcal{PF}}(C)$
$C \leftarrow$ infer(C)
**repeat**
    $C' \leftarrow C$;
    $C \leftarrow STEP(C)$;
**until** $C = C'$
**return** C
**end procedure**

---

L'algoritmo fa uso di due procedure: **infer** e **STEP**. La procedura infer è usata per aggiungere in modo automatico i vincoli **set, integer** e **pfun** al vincolo C per controllare che gli argomenti siano del tipo corretto. La procedura STEP rappresenta la parte principale di $SAT_{\mathcal{PF}}$ in quanto applica le procedure specializzate nel riscrivere i vincoli di C ritornando il vincolo modificato. L'esecuzione di STEP è iterata finchè non è raggiunto un punto fisso. Quando non possono essere applicate regole di riscrittura al vincolo considerato allora la procedura termina immediatamente e il constraint store rimane inalterato. Se non vi sono regole di riscrittura per un vincolo allora vine ritornato un vincolo irriducibile come parte di quelli generati da $SAT_{\mathcal{PF}}$. In modo più preciso, se $X$ e $X_i$ sono variabili e $t$ è un termine, i seguenti $\mathcal{PF} - constraint$ sono da considerarsi irriducibili:

- dom$(X_1, X_2)$, dove $X_1$ e $X_2$ sono variabili distinte
- ran$(X, t)$, dove $t$ è distinto da $X$ e $t$ non è l'insieme vuoto
- comp$(X_1, t, X_2)$ o comp$(t, X_1, X_2)$, dove $t$ non è l'insieme vuoto
- pfun$(X)$ senza il vincolo integer(X) in C

La presenza di questi vincoli non garantisce la soddisfacibilità di C. Il solver in questo modo risulta incompleto ovvero se ritorna un fallimento allora C è sicuramente non soddisfacibile ma non vale il vice versa.

# Capitolo 2

# JSetL

JSetL è una libreria Java che combina il paradigma di programmazione ad oggetti con alcune caratteristiche di programmazione a vincoli come variabili logiche, liste,unificazione e non determinismo [6].

L'unificazione può avvenire tra variabili logiche, liste e insiemi. I vincoli includono le operazioni di base sugli insiemi e sugli interi. La risoluzione dei vincoli sugli interi utilizza le tecniche di risoluzione su *domini finiti* (FD) mentre per gli insiemi vengono usate le tecniche di risoluzione su *insiemi finiti*(FS), se gli insiemi sono di interi e completamente specificati, altrimenti le procedure di risoluzione CLP(SET) per insiemi di qualsiasi tipo.
Il non determinismo è sfruttato sia per la ricerca delle soluzioni che per i vincoli su insiemi come unificazione e unione. Infine JSetL permette all'utente di definire nuovi vincoli e trattarli come se fossero built-in.

## 2.1 Variabili logiche

Le variabili logiche sono variabili che contengono valori non modificabili. Alle variabili possono essere associati valori attraverso relazioni (o *vincoli*) tra altre variabili o domini specifici. Quando il dominio di una variabile è ristretto ad un solo valore quella variabile è detta *bound*, ovvero legata a quel valore, altrimenti è detta *unbound*. Il vincolo di uguaglianza in particolare permette di associare un valore specifico alla variabile logica.
Le variabili logiche possono avere un *nome esterno* a cui sono associate. Il nome esterno è una stringa utile ad identificare la variabile nei vincoli in cui è coinvolta.

In JSetL una variabile logica è un'istanza della classe `LVar`. Questa classe fornisce i costruttori per creare variabili logiche e un insieme di metodi per manipolarli come i vincoli di uguaglianza e di appartenenza insiemistica. La

classe `LVar` ha un insieme di sottoclassi che specificano il tipo dei loro valori. Una prima distinzione si ha tra valori *atomici* e *strutturati*. Le variabili che possono avere valori atomici sono le *variabili logiche intere*(classe `IntLVar`) e gli *insiemi di variabili intere*(classe `SetLVar`) i cui valori sono rispettivamente numeri interi e insiemi di interi. Le variabili che invece assumono valori strutturati sono gli *insiemi di variabili logiche* e le *liste di variabili logiche* (classe `LSet` e classe `LList`) i cui valori sono rispettivamente insiemi e liste parzialmente specificati di elementi di qualsiasi tipo.

### 2.1.1 Classe LVar

Una variabile logica in JSetL è un'istanza della classe `LVar`, di seguito verranno mostrati i costruttori e alcuni metodi della classe.

**Costruttori**

- `LVar()`: crea una variabile logica unbound con nome esterno `?`
- `LVar(String extName)`: crea una variabile logica unbound con nome esterno `extName`
- `LVar(Object o)`: crea una variabile logica bound con nome esterno `?` e valore `Object o`
- `LVar(String extName, Object o)`: crea una variabile logica con nome esterno `extName` e valore `Object o`
- `LVar(LVar lv)`: crea una variabile logica bound con lo stesso valore di lv e nome `?`. Equivale a risolvere il vincolo `x.eq(lv)`
- `LVar(String extName, LVar lv)`: crea una variabile logica bound con lo stesso valore di lv e nome esterno `extName`.

**Vincoli su LVar**

- `Constraint eq(Object o)`: ritorna il vincolo atomico `this = o`, che unifica la variabile logica che invoca il metodo con `Object o`. In particolare se la variabile è unbound viene legata al valore di `o`.
- `Constraint neq(Object o)`: ritorna il vincolo atomico `this` $\neq$ `o`. Per soddisfare il vincolo è richiesto che le due variabili logiche confrontate siano diverse.

- `Constraint in(LSet ls)`: ritorna il vincolo atomico `this` $\in$ `ls`. Quando risolto il vincolo unificherà nondeterministicamente la variabile logica con ogni elemento dell'insieme `ls`. Il vincolo si dice risolto se almeno un'unificazione ha successo.

- `Constraint nin(LSet ls)`: ritorna il vincolo atomico `this` $\notin$ `ls`. Per risolvere il vincolo è richiesto che la variabile non appartenga all'insieme $ls$.

## 2.2 Insiemi logici

Un *insieme logico* è un particolare tipo di variabile logica il cui valore è una coppia $\langle elems, rest\rangle$ dove $elems$ è uni insieme $\{e_0, \ldots, e_n\}$, $n \geq 0$ di oggetti di tipo arbitrario, il valore dell'insieme, e $rest$ può essere sia l'insieme vuoto, sia un insieme logico unbound rappresentante il resto dell'insieme. Quando $rest$ è unbound si dice che l'insieme $s$ è *aperto* e si usa la notazione $\{e_0, \ldots, e_n|r\}$ per denotarlo, per contro se $rest$ è l'insieme vuoto si dice che $s$ è *chiuso* rappresentandolo con la notazione $\{e_0, \ldots, e_n\}$. Quando $elem$ è l'insieme vuoto e $rest$ è `null` l'insieme $s$ rappresenta l'insieme vuoto denotato da {}. In un insieme logico valgono le stesse proprietà degli insiemi della matematica, per cui l'ordine e le ripetizioni degli oggetti non conta, inoltre è possibile definire insiemi parzialmente specificati.

### 2.2.1 La classe LSet

In JSetL un insieme logico è definito come istanza della classe `LSet` che estende la classe `LCollection`. In particolare i valori dell'insieme sono istanze della classe `HashSet` che implementa l'interfaccia `Java.util.Set`. La classe fornisce i metodi per creare nuovi insiemi, a partire anche da insiemi già esistenti, e per assegnare dei valori all'insieme. In più gli insiemi logici possono essere usati per generare i vincoli che implementano le operazioni comuni per gli insiemi.

**Costruttori**

- `LSet()`: crea un'insieme logico unbound senza nome esterno

- `LSet(String extName)`: crea un'insieme logico unbound con nome esterno extName

- `LSet(Set<?> s)`: crea un insieme logico bound senza nome con valore s.

- `LSet(String extName, Set<?> s)`: crea un insieme logico bound con nome esterno extName e valore s.
- `LSet(LSet s)`: crea un insieme logico bound senza nome con valore s.
- `LSet(String extName, LSet s)`: crea un insieme logico bound con nome esterno extName e valore s.

**Metodi per creare LSet bound**

- `static LSet empty()`: ritorna l'insieme vuoto
- `LSet ins(Object o)`: se l'insieme logico è bound ritorna un nuovo insieme logico il cui valore è ottenuto aggiungendo l'elemento o agli elementi dell'insieme a cui è legato. Se l'insieme è unbound ritorna un nuovo insieme il cui valore è l'oggetto o e l'insieme stesso come resto.
- `LSet insAll(Object[] c)`: se l'insieme logico è bound ritorna un nuovo insieme logico il cui valore è ottenuto aggiungendo gli elementi di c agli elementi dell'insieme a cui è legato. Se l'insieme è unbound ritorna un nuovo insieme il cui valore è l'insieme di tutti gli elementi di c e l'insieme stesso come resto.
- `static LSet mkSet(int n)`: ritorna un insieme logico il cui valore è un insieme di n variabili logiche unbound
- `static LSet mkSet(String extName,int n)`: ritorna un insieme logico il cui valore è un insieme di n variabili logiche unbound e nome extName

L'ordine negli insiemi non è importante quindi non è necessario fornire metodi distinti per l'inserzione in testa e in coda perchè questi produrrebbero lo stesso risultato.

**Vincoli forniti dalla classe**

- `Constraint eq(Set<?> ls)`
- `Constraint eq(LSet ls)`: ritorna il vincolo atomico this = ls, che *unifica* l'insieme che invoca il metodo con ls. Se l'insieme è unbound e ls è bound all'insieme s, allora l'insieme viene legato al valore s dopo la risoluzione del vincolo. Se ls è unbound l'insieme rimane unbound.
- `Constraint neq(Set<?> ls)`
- `Constraint neq(LSet ls)`: ritorna il vincolo $this \neq ls$.

- `Constraint contains(LVar lv)`
- `Constraint contains(Object lv)`: ritorna il vincolo atomico $lv \in this$.
- `Constraint ncontains(LVar lv)`
- `Constraint ncontains(Object lv)`: ritorna il vincolo atomico $lv \notin this$.
- `Constraint diff(LSet ls1,LSet ls2)` ritorna il vincolo atomico $this = ls1 \setminus ls2$
- `Constraint disj(LSet ls)` ritorna il vincolo atomico $this||ls$
- `Constraint inters(LSet ls1,LSet ls2)` ritorna il vincolo atomico $this = ls1 \cap ls2$
- `Constraint less(LSet ls,LVar lv)` ritorna il vincolo atomico $this = less(ls, lv)$, dove $less(ls, lv)$ è una funzione che rimuove dall'insieme ls la variabile lv
- `Constraint size(Integer i)` ritorna il vincolo atomico $|this| = i$
- `Constraint subset(LSet ls)` ritorna il vincolo atomico $this \subseteq ls$
- `Constraint union(LSet ls1,LSet ls2)` ritorna il vincolo atomico $this = ls1 \cup ls2$

## 2.3 Vincoli

I vincoli sono le operazioni che possono essere applicate alle variabili logiche, insiemi logici e liste. Queste operazioni possono essere effettuate anche se gli oggetti coinvolti non hanno un valore specifico associato.
Un vincolo in JSetL è un'espressione che può assumere una delle seguenti forme:

- *vincoli atomici*
  - il *vincolo vuoto*, denotato `[]`
  - $e_0.op(e_1, ..., e_n)$ o $op(e_1, ..., e_n)$ con $n = 0, ..., 3$
    dove *op* è il nome del vincolo e $e_i (0 \leq i \leq 3)$ sono espressioni il cui tipo dipende da *op*. In particolare *op* può essere uno dei metodi predefiniti che implementano operazioni come l'uguaglianza, disuguaglianza, confronto tra interi, intersezione e unione.

- *vincoli composti*
  - $c_1$.`and`($c_2$) (*congiunzione*)
  - $c_1$.`or`($c_2$) (*disgiunzione*)
  - $c_1$.`impliesTest`($c_2$) (*implicazione*)

  dove $c_1$ e $c_2$ rappresentano vincoli di JSetL e `and, or, impliesTest` rappresentano le operazioni logiche di *congiunzione*($c_1 \wedge c_2$), *disgiunzione*($c_1 \vee c_2$) e *implicazione*($c_1 \rightarrow c_2$) tra $c_1$ e $c_2$ rispettivamente.

- *vincolo di negazione*
  - $c_1$.`notTest`() (negazione)

  dove $c_1$ rappresenta un vincolo di JSetL e `notTest` rappresenta la negazione di $c_1$ ($\neg c_1$).

I vincoli in JSetL sono definiti come istanze della classe `Constraint`. Gli oggetti `Constraint` sono creati attraverso i costruttori della classe corrispondente oppure come risultato dell'invocazione di alcuni metodi forniti dalle classi che implementano oggetti logici.

### 2.3.1 Classe Constraint

I vincoli in JSetL sono istanze della classe `Constraint`. Di seguito sono mostrati i costruttori e alcuni metodi della classe.

**Costruttori**

- `Constraint()`: crea il vincolo vuoto.
- `Constraint(String extName, Object o1, ..., Object` $o_n$`)` : crea il vincolo con nome `extName` e argomenti `o1`, ..., $o_n$, con $1 \leq n \leq 4$.

**Metodi**

- `Constraint and(Constraint c)`: ritorna il vincolo `this` $\wedge$ `c`
- `Constraint impliesTest(Constraint c)`: ritorna il vincolo `this` $\longrightarrow$ `c`, dove $\longrightarrow$ è l'implicazione logica
- `Constraint not()`: ritorna il vincolo $\neg$ `this`, dove $\neg$ è la negazione logica
- `Constraint or(Constraint c)`: ritorna il vincolo `this` $\vee$ `c`

## 2.4 Risolvere i vincoli

I vincoli vengono risolti attraverso un *risolutore di vincoli*. In JSetL un risolutore di vincoli può essere creato come istanza della classe `SolverClass`. La classe fornisce i metodi per aggiungere vincoli, controllarne la soddisfacibilità e trovare le possibili soluzioni. I vincoli vengono risolti dal risolutore per mezzo di riscritture. Una volta invocato il metodo `solve()` o `check()` il risolutore estrae dal *constraint store* un vincolo ancora non in forma risolta e vi applica una *regola di riscrittura* se disponibile, fatto ciò passa al vincolo successivo. La procedura termina quando tutti i vincoli sono in forma risolta o non ci sono più regole di riscrittura da applicare, oppure viene lanciata un'eccezione di tipo `Failure`, nel caso di vincolo non soddisfacibile.

### 2.4.1 Classe SolverClass

I risolutori di vincoli sono istanze della classe `SolverClass`. Di seguito sono mostrati il costruttore e alcuni metodi della classe.

**Costruttore**

- `SolverClass()`: crea un risolutore di vincoli

**Metodi**

- `void add(Constraint c)`: aggiunge un vincolo al 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 risolutore e il vincolo `c` da aggiungere siano soddisfacibili. Se possibile viene calcolata anche una soluzione e aggiunto il vincolo al risolutore. Se il vincolo non è soddisfacibile i vincoli nel solver rimangono invariati e il vincolo `c` scartato
- `void solve()`: risolve i vincoli nel risolutore. Come il metodo `check()` cerca una soluzione se possibile, altrimenti viene lanciata un'eccezione di tipo `Failure`

## 2.5 Dettagli sull'implementazione dei vincoli

Un vincolo è un'istanza della classe `Constraint`. La classe si caratterizza dall'essere un contenitore di oggetti di tipo `AConstraint` con i relativi me-

todi per gestirli. Un `AConstraint` rappresenta il modello con cui vengono tenuti in memoria i vincoli atomici. Un oggetto di questa classe è costituito da un attributo `cons` che contiene il codice univoco del vincolo, quattro attributi `arg1,arg2,arg3 e arg4` che rappresentano gli argomenti del vincolo e l'attributo `solved` che indica se il vincolo è in forma risolta oppure no.

### 2.5.1 Utilizzo dei vincoli nella libreria

#### Uso nel metodo SolverClass::add(Constraint c)

Il metodo `add()` prende come argomento un vincolo e lo aggiunge all'oggetto che contiene tutti i vincoli aggiunti al solver fino a quel momento.

#### Uso nel metodo SolverClass::solve()

Il metodo `solve()` scorre la lista dei vincoli e risolve quelli che non sono in forma risolta, se un vincolo è inconsistente viene lanciata un'eccezione.

#### Uso nei metodi di vincolo

Il metodo genera un nuovo oggetto di tipo Constraint passando come parametri l'oggetto su cui è invocato il metodo e i parametri ausiliari. Un' esempio di questi metodi è `in(LSet z)` della classe `LVar`.

### 2.5.2 Aggiungere un nuovo vincolo built-in

Aggiungere un nuovo vincolo vuol dire creare il codice univoco che lo identifica, specificare le regole di riscrittura con cui il solver possa risolverlo e infine identificare la classe in cui utilizzarlo.

#### Generare il codice univoco del nuovo vincolo

Per generare il codice del vincolo deve essere modificata la classe `Environment` aggiungendo la variabile statica che lo contiene. La classe `Environment` genera i codici invocando il metodo
`Environment::getMethodCode(Boolean b)` che passa come parametro la costante `true` per indicare che sono built-in. In seguito viene impostato nella lista `constraintNames[constraintCode]` nella posizione constraintCode il nome del vincolo e infine nel metodo `Environment::toString()` inserire alla fine della lista di `else if` il modo in cui tradurre in stringa il vincolo.

**Esempio sul codice:**

```
public class Environment{

     .
     .
     .

     //Dichiarare prima la variabile relativa al nuovo vincolo
     static int newConstraintCode;


     static {

          constraintsCode = firstConstraintsCode;

          .
          .
          .
          notBoolCode    = getMethodCode(true);
          andBoolCode    = getMethodCode(true);
          orBoolCode     = getMethodCode(true);
          impliesBoolCode = getMethodCode(true);
          iffBoolCode    = getMethodCode(true);

          // Nuovo codice univoco
    -->   newConstraintCode = getMethodCode(true);

          .
          .
          .
          constraintNames[andBoolCode] = "andBool";
          constraintNames[orBoolCode]  = "orBool";
          constraintNames[impliesBoolCode]= "impliesBool";
          constraintNames[iffBoolCode] = "iffBool";
          constraintNames[notBoolCode] = "notBool";

          // Nome del vincolo
     -->  constraintNames[newConstraintCode] = "newConstraintName";

     }
```

```
      .
      .
      .

     protected static String toString(int cons, Object arg1,
                                      Object arg2, Object arg3,
                                      Object arg4){

       if (cons == domCode)
           return arg1.toString() + "::" + arg2.toString();
       else if (cons == labelCode)
           return arg1.toString() + ".label()";

       else if (cons == eqCode)
           return arg1.toString() + " = " + arg2.toString();
       else if (cons == neqCode)
           return arg1.toString() + " neq " + arg2.toString();

       .
       .
       .

       else if (cons == newConstraintCode)
-->       return <*TRADUZIONE IN STRINGA DEL VINCOLO*> ;

       // other constraints
       else {
           String out = code_to_name(cons) + "(";
           if (arg1 != null) out += arg1.toString();
           if (arg2 != null) out += "," + arg2.toString();
           if (arg3 != null) out += "," + arg3.toString();
           if (arg4 != null) out += "," + arg4.toString();
           out += ")";
           return out;
       }
   }

}
```

### Aggiungere le regole di riscrittura per il nuovo vincolo

Le regole di riscrittura devono essere aggiunte nelle apposite classi `RwRules`. Vi sono attualmente sei di queste classi:

- `RwRulesEq`: specifica le regole di riscrittura per i test di uguaglianza
- `RwRulesBool`: specifica le regole di riscrittura per gli operatori logici
- `RwRulesSet`: specifica le regole di riscrittura per operatori insiemistici
- `RwRulesFD`: specifica le regole di riscrittura per oggetti con dominio finito
- `RwRulesFS`: specifica le regole di riscrittura per insiemi finiti
- `RwRulesMeta`: specifica le regole di riscrittura per alcune operazioni tra vincoli

Queste classi estendono `LibConstraintsClass`. Per aggiungere le regole di riscrittura serve aggiungere al metodo `RwRules::solveConstraint(AConstraint s)` la decodifica del codice del constraint con la chiamata al metodo specifico di riscrittura. Il metodo di riscrittura prende come argomento un `AConstraint s` e a seconda dei tipi degli argomenti di `s` invoca uno specifico metodo di riscrittura.

### Esempio sul codice:

```
public class RwRules extends LibConstraintsClass {

    protected boolean solveConstraint(AConstraint s) throws Failure
        {
        if (s.cons == Environment.complCode)
               compl((SetLVar) s.arg1, (SetLVar) s.arg2, s);
        else if (s.cons == Environment.disjCode)
               disj(s);
        else if (s.cons == Environment.subsetCode)
               subset(s);
        .
        .
        .
        //Decodifica del constraint
        else if (s.cons == Environment.newConstraintCode)
```

```
                newConstraint((SetLVar) s.arg1, (SetLVar) s.arg2,
                              (SetLVar) s.arg3, s,(SetLVar) s.arg4);
        else
                return false;

    }

    protected void newConstraint(SetLVar arg1, SetLVar arg2,
                                 SetLVar arg3, SetLVar arg4){

    /*
      CODICE PER GESTIRE LA RISCRITTURA
    */

    }
}
```

### Uso del nuovo vincolo

Per usare il nuovo vincolo si aggiunge alla classe desiderata un metodo che restituisce un oggetto di tipo `Constraint` che lo genera.

### Esempio sul codice:

```
class MiaClasse{

   .
   .
   .

   //I parametri dipendono dal tipo di vincolo implementato
   public Constraint newConstraint(Object z){
      return new Constraint(this,Environment.newConstraintCode,z);
   }

   .
   .
   .
}
```

# Capitolo 3

# Rappresentazione delle funzioni parziali

La rappresentazione delle funzioni parziali in JSetL rispecchia la definizione data in precedenza. Le funzioni parziali vengono quindi trattate come particolari insiemi i cui elementi sono coppie ordinate rappresentate dalla classe `LMap`. La classe `LMap` estende `LSet`, dalla quale eredita in particolare i vincoli associati. In questo modo una `LMap` è un insieme logico con la particolarità che i suoi elementi sono oggetti della classe `Pair`.

## 3.1 Creare una LMap

Secondo la specifica una funzione parziale può essere solo nella forma:

- $r = \{\}$
- $r = \{[x_1, t_1], [x_2, t_2], \ldots, [x_n, t_n]\}$
- $r = \{[x_1, t_1], [x_2, t_2], \ldots, [x_n, t_n] | s\}$

per cui la classe è stata fornita solo dei metodi per generare funzioni con quella forma in particolare attraverso l'utilizzo dei metodi `empty()`, ereditato da `LSet()` che genera un' insieme vuoto, e `ins()`, che inserisce nuovi elementi alla funzione.

La classe `Pair` rappresenta un'astrazione per le coppie. Questi oggetti sono particolari variabili logiche della forma $\langle key, value \rangle$ dove *key* rappresenta l'oggetto che identifica la coppia e *value* il valore dato. Sulle coppie valgono tutti i vincoli e le proprietà per le variabili logiche in quanto estendono la classe `LVar`, in particolare possono non essere legate ad un valore

ovvero *unbound*. Per le coppie inoltre sono stati forniti il vincolo di uguaglianza tra coppie e il vincolo di appartenenza. Il vincolo di uguaglianza segue la seguente definizione: siano $p_1$ e $p_2$ due oggetti di tipo Pair allora $p_1 = p_2 \iff p_1.first = p_2.first \wedge p_1.second = p_2.second$. Il vincolo di appartenenza si differenzia da quello implementato nella libreria in quanto usa l'uguaglianza tra coppie appena definita. Di seguito un esempio:

```
/** Genero la funzione vuota emptyFunction = {} **/
LMap emptyFunction = LMap.empty();

/** Genero la funzione chiusa closeFun = {[ob1,ob2],[ob3,ob4]} **/
LMap closeFun = LMap.empty.ins(new Pair(ob1,ob2))
          .ins(new Pair(ob3,ob4));

/** Genero la funzione aperta
    openFun = {[ob1,ob2],[ob3,ob4] | s}**/
LMap openFun = new LMap().ins(new Pair(ob1,ob2)).ins(new
    Pair(ob3,ob4));
```

Si noti come per differenziare le funzioni chiuse da quelle aperte si utilizzi come primo elemento inserito l'insieme vuoto.

## 3.2 Vincoli su LMap

La classe eredita i vincoli sugli insiemi da `LSet` e in più fornisce i vincoli specifici per le funzioni parziali per cui si possono distinguere tre tipi di vincoli:

- Vincoli insiemistici: sono i vincoli sugli insiemi ereditati dalla classe `LSet`, di questi fanno parte `in()`, `eq()`, `inters()`, `union()` e `disj()`
- Vincoli primitivi su funzioni parziali: sono i vincoli primitivi `dom()`, `ran()`, `pfun()` e `comp()` definiti nel linguaggio CLP($\mathcal{PF}$) e direttamente implementati nella libreria `JSetL`
- Vincoli composti su funzioni parziali: sono i vincoli ottenuti dalla congiunzione dei vincoli primitivi e insiemistici come `dres()` e `dran()`

In generale un vincolo viene generato dalla classe attraverso il metodo omonimo che segue la struttura:

```
public Constraint nomeVincolo(Object ob1,..){

    Constraint c = new Constraint(...);
     .
     .
    return c;

}
```

Nella classe `LMap` i vincoli primitivi su funzioni parziali sono così implementati:

- `Constraint dom(LSet d)`: il metodo prende come argomento un oggetto di tipo `LSet d` che rappresenta il dominio della funzione e restituisce l'oggetto `Constraint` rispettivo.
- `Constraint ran(LSet r)`: il metodo prende come argomento un oggetto di tipo `LSet r` che rappresenta il codominio della funzione e restituisce l'oggetto `Constraint` rispettivo.
- `Constraint pfun()`: il metodo restituisce l'oggetto `Constraint` rispettivo.
- `Constraint comp(LMap r, LMap q)`: il metodo prende come argomenti gli oggetti di tipo `LMap r` e `q`, che rappresentano rispettivamente la funzione con qui deve essere effettuata la composizione e la funzione composizione di `this` e `r`, e restituisce l'oggetto `Constraint` rispettivo.

Di seguito è mostrato come esempio l'implementazione di `dom()`:

```
public Constraint dom(LSet dom){
    return new Constraint(this,Environment.mapDomCode,dom);
}
```

Nel codice si può notare che come parametri del costruttore vengono passati gli argomenti del vincolo e il codice che lo identifica.
La classe `LMap` fornisce anche i vincoli composti su funzioni parziali di cui alcuni così definiti:

- `Constraint dres(LMap s, LSet a)`: il metodo prende come argomento un oggetto di tipo `LSet a` che rappresenta il dominio ristretto

e un oggetto di tipo `LMap s` che rappresenta la funzione ristretta restituendo un oggetto di tipo `Constraint`. Nel metodo vengono generati dei vincoli intermedi che congiunti producono il vincolo dres.

- `Constraint ndres(LMap s, LSet a)`: il metodo prende come argomento un oggetto di tipo `LSet a` che rappresenta l'antirestrizione del dominio e un oggetto di tipo `LMap s` che rappresenta la funzione ristretta restituendo un oggetto di tipo `Constraint`. Nel metodo vengono generati dei vincoli intermedi che congiunti producono il vincolo ndres.
- `Constraint id()`: il metodo restituisce un oggetto di tipo `Constraint` creato dalla congiunzione dei vincoli di dominio, codominio e composizione sull'oggetto che ha chiamato il metodo.

Di seguito è mostrato come esempio l'implementazione di `dres()`:

```
public Constraint dres(LMap s,LSet a){
      LSet dr = new LSet();
      LSet ds = new LSet();

      Constraint dom1 = new Constraint(this,
          Environment.mapDomCode, dr);
      Constraint dom2 = new Constraint(s, Environment.mapDomCode,
          ds);
      Constraint inters = a.inters(dr, ds);
      Constraint subs = s.subset(this);

      return dom1.add(dom2).add(inters).add(subs);
}
```

### 3.2.1 Metodi ausiliari

La classe fornisce i metodi ausiliari `isGround()` e `isPfun()` usati in particolare nell'implementazione delle regole di riscrittura e disponibili all'utente.

#### isGround()

Il metodo controlla che tutti gli elementi del dominio della funzione siano legati ad un valore, ovvero *bound*. Di seguito il codice con cui è stato implementato.

```
public static boolean isGround(LMap r){
       Pair p;
       LMap rest = r;

       if(!r.isInit()){
         return false;
       }

       if(!r.isClosed()){
         return false;
       }

     while(rest.getSize() >= 1){
       p = new Pair((Pair)rest.getOne());
       rest = (LMap)rest.removeOne();

       if(!p.isFirstInit()){
         return false;
       }

     }

     return true;
   }
```

Nel codice si osserva che la funzione unbound e la funzione aperta non siano per definizione ground.

### isPfun()

Il metodo controlla che sia verificato il vincolo di funzione parziale su funzioni ground. Di seguito il codice con cui è stato implementato.

```
public static boolean isPfun(LMap p){
     Pair p1;
     Pair p2;

     if(isGround(p)){
       for(int y1 = 0 ; y1 < p.getSize();++y1){
         for(int y2 = y1+1; y2 < p.getSize();++y2){
           p1 = new Pair((Pair)p.get(y1));
```

```
          p2 = new Pair((Pair)p.get(y2));

          if(p1.getFirst().equals(p2.getFirst())){
            return false;
          }
        }
      }
    }else{
      throw new NotDefDomain();
    }

    return true;
  }
```

# Capitolo 4

# Regole di riscrittura per vincoli su funzioni parziali

## 4.1 Regole di riscrittura

Per ogni simbolo di vincolo $\pi \in \Pi_C$, si definisce una *regola di riscrittura* specifica per quel tipo di vincolo. Ogni procedura di risoluzione applica ripetutamente al vincolo di input C una collezione di *regole di riscrittura* per $\pi$ finché il vincolo C non viene riscritto in **false** o nessuna altra regola per $\pi$ si applica a C. In ogni momento C rappresenta il *constraint store* manipolato dal risolutore.
Le regole di riscrittura hanno la seguente forma:

$$\frac{precondizioni}{\{C_1,C_2,...,C_n\}\rightarrow\{C'_1,C'_2,...,C'_n\}}$$

dove $C_i$ e $C'_i$ sono $(\mathcal{SET}-\mathcal{PF})$-constraint primitivi e le *precondizioni* sono condizioni booleane, possibilmente vuote, sui termini che occorrono in $C_1,\ldots,C_n$. Per poter applicare la regola ogni precondizione deve essere soddisfatta. $\{C_1,\ldots,C_n\}\rightarrow\{C'_1,C'_2,\ldots,C'_n\}$ $(n,m\geq 0)$ rappresenta la modifica al constraint store causata dall'applicazione della regola.
Nelle regole sono usate le seguenti convenzioni e predicati ausiliari:

- $\mathcal{V}$ rappresenta l'insieme delle variabili
- $empty(s)$ è un predicato ausiliario definito come segue: $s=\emptyset\vee(s=in(x,y)\wedge x>y)$ (si nota che, $\neg empty(s)$ è verificata anche se s è una variabile unbound)
- $is\_ground(t)$ è un predicato ausiliario che è vero quando il termine t non contiene alcuna variabile unbound

- $is\_pfun(r)$ è un predicato ausiliario definito come segue:
  $is\_pfun(r) = true \iff \forall x, y_1, y_2 : ([x, y_1] \in r \wedge [x, y_2] \in r \rightarrow y_1 = y_2)$
  $is\_pfun(r)$ è usato per verificare che una relazione r, con dominio ground, è una funzione parziale

### 4.1.1 Regole non deterministiche

Per alcuni vincoli primitivi esistono regole di riscrittura diverse che necessitano delle medesime precondizioni per essere applicate. Queste vengono dette regole non deterministiche in quanto in un certo momento della computazione per il simbolo $\pi$ possono essere applicate tutte queste regole portando ad avere altrettante possibili soluzioni valide.
La forma generale di una regola non deterministica è la seguente:

$$\frac{precondizioni}{\{C_1, C_2, \ldots, C_n\} \rightarrow \{C_1', C_2', \ldots, C_n'\} \quad o \quad \{C_1'', C_2'', \ldots, C_n''\}}$$

dove $\{C_1', \ldots, C_2'\}$ e $\{C_1'', \ldots, C_2''\}$ sono le modifiche fatte in modo non deterministico al constraint store.

## 4.2 Vincolo di dominio

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

$$\frac{r \in \mathcal{V}}{\{dom(r,r)\} \rightarrow \{r = \emptyset\}} \tag{4.1}$$

$$\frac{empty(a)}{\{dom(r,a)\} \rightarrow \{r = \emptyset\}} \tag{4.2}$$

$$\frac{empty(r)}{\{dom(r,a)\} \rightarrow \{a = \emptyset\}} \tag{4.3}$$

$$\frac{r = \{[x,y]|rr\} \qquad \neg empty(a)}{\{dom(r,a)\} \rightarrow \{a = \{x|rs\}, [x,y] \text{ nin } rr, dom(rr,rs)\}} \tag{4.4}$$

$$\frac{r \in \mathcal{V} \qquad a = \{x|rs\}}{\{dom(r,a)\} \rightarrow \{r = \{[x,y]|rr\}, x \text{ nin } rs, dom(rr,rs)\}} \tag{4.5}$$

Le regole sono definite per ricorsione. I primi tre casi rappresentano i casi base mentre gli ultimi due i casi ricorsivi.

### 4.2.1 Implementazione in JSetL

In JSetL l'uso del vincolo si traduce nell'invocare il metodo `dom(LSet l)` che equivale a $dom(this, l)$. Le regole sono implementate dal metodo `mapDomainRule()` con il seguente codice:

```
private void mapDomainRule(LSet a,LMap r, AConstraint s) throws
    Failure{
   /** Le condizioni degli statement if verificano le
      precondizioni**/

   /** CASI BASE **/
   /** Implementazione delle regole 4.1 e 4.2 **/
  if(a.equals(r) || (a.isInit() && a.isEmpty())){

    s.arg2 = LSet.empty();
    s.cons = Environment.eqCode;
    Solver.storeInvariato = false;
    return;
  }

    /** Implementazione della regola 4.3 **/
  if(r.isInit() && r.isEmpty()){
    s.arg1 = LMap.empty();
```

```
    s.cons = Environment.eqCode;
  Solver.storeInvariato = false;
  return;
}

/** CASI RICORSIVI**/
  /** Implementazione della regola 4.4 **/
  /** La regola controlla che il dominio sia corretto
      data una funzione r inizializzata **/
if(r instanceof LMap && r.isInit()){
  Pair tmp = (Pair)r.getOne();
  LMap rest = (LMap)r.removeOne();

  LSet rs = new LSet();
  Object t = tmp.getFirst();

  LSet tmpDom;

  if(t != null){
    tmpDom = rs.ins(t);
  }else{
    tmpDom = rs.ins(new LVar());
  }


  Solver.add(new AConstraint(a,Environment.eqCode,tmpDom));

  s.arg1 = rest;
  s.arg2 = rs;
  mapDomain(s);
  Solver.storeInvariato = false;
  return;
}

  /** Implementazione della regola 4.5 **/
  /** La regola controlla che la funzione sia corretta dato
      il dominio a inizializzato **/
if(a instanceof LSet && a.isInit()){
  Object tmp = a.getOne();
  LSet rest = a.removeOne();

  LMap rs = new LMap();
  LMap tmpMap = rs.ins(new Pair(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;
        mapDomain(s);
    Solver.storeInvariato = false;
    return;
  }

}
```

Il metodo viene invocato dopo aver selezionato gli argomenti corretti con il metodo `domain()` della classe `RwRuleFD` che controlla gli argomenti del vincolo e seleziona i valori equivalenti tramite i campi `equ`, qualora fosse stato invocato in precedenza il vincolo `eq` su uno degli argomenti.

```
private void domain(LSet l,LMap lmap, AConstraint s) throws
    Failure{

     if(lmap.equ == null)
       if(l.equ == null){
          mapDomainRule(l,lmap,s);
       }else
        domain((LSet)l.equ,lmap,s);
     else
       domain(l,(LMap)lmap.equ,s);

}
```

Nel metodo `mapDomainRule()` ogni istruzione if coincide con una regola di riscrittura specifica, in particolare la condizione dell'if è una traduzione delle precondizioni della regola. I metodi proposti in questo capitolo sono forniti dalla classe `RwRuleFD`.

### 4.2.2 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo del vincolo in alcuni programmi d'esempio.

**Esempio 1:**

Sia data la funzione $f = \{[1, a], [2, b], [3, c]\}$ si vuole calcolare il dominio della funzione:

```
public static void main() throws Failure{

   /** Viene creato il solver **/
   SolverClass solver = new SolverClass();

   /** Viene creata la funzione f chiusa **/
   LMap f = LMap.empty().ins(new Pair(1,'a')).ins(new
       Pair(2,'b')).ins(new Pair(3,'c'));

   /** Viene dichiarato l'insieme che rappresenta il dominio **/
   LSet dominio = new LSet();

   /** Viene aggiunto il vincolo al solver **/
   solver.add(f.dom(dominio));

   /** Risoluzione del vincolo **/
   solver.solve();

   /** Stampa del risultato **/
   dominio.setName("Dominio");
   dominio.output();

}
```

Il metodo `output()` stampa il valore della funzione, in questo caso dopo la risoluzione l'output è il seguente:

| Dominio={1,2,3} |
|---|

**Esempio 2:**

Si vogliono verificare i casi base del vincolo dom:

```
public static void main() throws Failure{

   /** Viene creato il solver **/
   SolverClass solver = new SolverClass();

   System.out.println("Caso base 1: r non inizializzato V e
       dom(r,r)");

   LMap r = new LMap();

   solver.add(r.dom(r));
   solver.solve();

   r.setName("r");
   r.output();

   solver.clearStore();

   System.out.println("Caso base 2: dom(r,{})");


   LSet a = LSet.empty();

   solver.add(r.dom(a));
   solver.solve();


   r.setName("r");
   r.output();
```

```
    a.setName("a");
    a.output();

    solver.clearStore();

    System.out.println("Caso base 3: dom({},a)");


    r = LMap.empty();

    solver.add(r.dom(a));
    solver.solve();

    r.setName("r");
    r.output();

    a.setName("a");
    a.output();

}
```

Il programma usa il metodo `clearStore()` che cancella tutti i vincoli presenti nel solver. L'output del programma è il seguente:

```
Caso base 1: r non inizializzato e dom(r,r)
r = {}

Caso base 2: dom(r,{})
r = {}
a = {}

Caso base 3: dom({},a)
r = {}
a = {}
```

### Esempio 3

Sia dato l'insieme $a = \{b, c, d\}$, si vuole costruire una funzione f tale che il suo dominio sia l'inseme $a$:

```
public static void main() throws Failure{

   /** Viene creato il solver **/
   SolverClass solver = new SolverClass();

   /** Viene creato l'insieme a chiuso **/
   LSet a = new LSet("domRest").ins('b').ins('c').ins('d');

   /** Viene dichiarata la funzione **/
   LMap f = new LMap("f");

   /** Viene aggiunto il vincolo al solver **/
   solver.add(f.dom(a));

   /** Risoluzione del vincolo **/
   solver.solve();

   /** Stampa del risultato **/
   f.output();

}
```

L'output del programma è il seguente:

$$f = \{[b, \_?], [c, \_?], [d, \_?]\}$$

**Esempio 4:**

Sia dato l'insieme parzialmente specificato $a = \{1, 2, 3|b\}$ si vuole trovare per quali vincoli $f$ sia la funzione il cui dominio è l'insieme $a$.

```
public static void main() throws Failure{

   /** Viene creato il solver **/
   SolverClass solver = new SolverClass();

   /** Viene creato l'insieme a aperto **/
   LSet a = new LSet("b").ins(1).ins(2).ins(3).setName("a");
```

```
    /** Viene dichiarata la funzione **/
    LMap f = new LMap("f");

    /** Viene aggiunto il vincolo al solver **/
    solver.add(f.dom(a));

    /** Risoluzione del vincolo **/
    solver.solve();

    /** Stampa del risultato **/
    a.output();
    f.output();
    solver.showStore();
}
```

L'output del programma è il seguente:

$a = \{1, 2, 3|\_b\}$
$f = \{[1, \_?], [2, \_?], [3, \_?]|\_?\}$
$Store : mapDom(\_?, \_b)$ AND 3 nin $\_b$ AND 2 nin $\_b$ AND 1 nin $\_b$

I vincoli che il solver deve ancora risolvere sono quelli che permettono di mantenere le proprietà di funzione parziale senza l'utilizzo esplicito del vincolo pfun di cui si parlerà in seguito.

## 4.3 Vincolo di codominio

Il vincolo di codominio nel linguaggio CLP($\mathcal{PF}$) ha la forma $ran(r, b)$ dove $r$ è una funzione parziale e $b$ un insieme. Il vincolo è soddisfatto se e solo se l'insieme $b$ è il codominio della funzione $r$.

Le regole per il vincolo sono le seguenti:

$$\frac{r \in \mathcal{V}}{\{ran(r,r)\} \rightarrow \{r = \emptyset\}} \tag{4.6}$$

$$\frac{empty(b)}{\{ran(r,b)\} \rightarrow \{r = \emptyset\}} \tag{4.7}$$

$$\frac{empty(r)}{\{ran(r,b)\} \rightarrow \{b = \emptyset\}} \tag{4.8}$$

$$\frac{r = \{[x,y]|rr\} \qquad \neg empty(b)}{\{ran(r,b)\} \rightarrow \{b = \{y|rs\}, [x,y] \text{ nin } rr, ran(rr,rs)\}} \tag{4.9}$$

### 4.3.1 Implementazione in JSetL

In JSetL l'uso del vincolo si traduce nell'invocare il metodo `ran(LSet l)` che equivale a $ran(this, l)$. Le regole sono state implementate dal metodo `mapRangeRule()` con il seguente codice:

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

    /** CASI BASE **/
    /** Implementa le regole 4.6 e 4.7**/
    if(a.equals(r) || a.isInit() && a.isEmpty()){
    s.arg2 = LSet.empty();
    s.cons = Environment.eqCode;
    Solver.storeInvariato = false;
    return;
  }

    /** Implementa la regola 4.8 **/
  if(r.isInit() && r.isEmpty()){
    s.arg1 = LMap.empty();
    s.cons = Environment.eqCode;
    Solver.storeInvariato = false;
    return;
  }
```

```
      /** CASO RICORSIVO **/
      /** Implementa la regola 4.9 **/
      /** La regola controlla che il codominio sia corretto
          data una funzione parziale inizializzata **/
    if(r instanceof LMap && r.isInit()){
      Pair tmp = (Pair)r.getOne();
      LMap rest = (LMap)r.removeOne();

      LSet rs = new LSet();
      LSet tmpRange = rs.ins(tmp.getSecond());

      Solver.add(new AConstraint(a,Environment.eqCode,tmpRange));
      Solver.add(new AConstraint(tmp,Environment.ninCode,rest));
      s.arg1 = rest;
      s.arg2 = rs;
      range(s);
      Solver.storeInvariato = false;
      return;
    }

  }
```

Anche in questo caso, e nei successivi, è usata una funzione ausiliaria per la selezione degli argomenti e le condizioni dell'if come verifica delle precondizioni.

### 4.3.2 Esempi d'uso

Di seguito verranno mostrati degli esempi di utilizzo del vincolo in alcuni programmi.

**Esempio 1:**

Si vogliono verificare i casi base del vincolo ran:

```
/** Viene creato il solver **/
SolverClass solver = new SolverClass();

System.out.println("Caso base 1: r non inizializzato e
    ran(r,r)");
```

```
    LMap r = new LMap();

    solver.add(r.ran(r));
    solver.solve();

    r.setName("r");
    r.output();

    solver.clearStore();

    System.out.println("Caso base 2: ran(r,{)}");


    LSet b = LSet.empty();

    solver.add(r.ran(b));
    solver.solve();


    r.setName("r");
    r.output();

    b.setName("b");
    b.output();

    solver.clearStore();

    System.out.println("Caso base 3: ran({},b)");


    r = LMap.empty();

    solver.add(r.ran(b));
    solver.solve();

    r.setName("r");
    r.output();

    b.setName("b");
    b.output();

}
```

L'output del programma è il medesimo dell'esempio 2 del vincolo di dominio, escluso il nome delle variabili.

**Esempio 2:**

Sia dato l'insieme $b = \{1\}$ si vuole trovare la funzione f associata:

```
public static void main() throws Failure{

   /** Viene creato il solver **/
   SolverClass solver = new SolverClass();

   /** Viene creata l'insieme b chiuso **/
   LSet b = LSet.empty().ins(1);

   /** Viene dichiarata la funzione **/
   LMap f = new LMap("f");

   /** Viene aggiunto il vincolo al solver **/
   solver.add(f.ran(b));

   /** Risoluzione del vincolo **/
   solver.solve();

   /** Stampa del risultato **/
   f.output();
   solver.showStore();

}
```

Il metodo `showStore()` stampa i vincoli ancora non risolti del solver. L'output del programma è il seguente:

```
f = unknown
Store : ran(_f,{1})
```

Le funzione f non viene generata in quanto il vincolo $ran(f, \{1\})$ è irriducibile. Un vincolo si dice irriducibile se non può essere riscritto da alcuna regola di riscrittura. In questo caso le possibili soluzioni sono infinite (es. $f = \{[\_?, 1]\}$, $f = \{[\_?, 1], [\_?, 1]\}$,..., $f = \{[\_?, 1], \ldots, [\_?, 1]\}$). Una congiun-

zione di vincoli con un vincolo irriducibile non sempre è soddisfacibile, un esempio è il seguente: $ran(X, \{1\}) \wedge ran(X, \{A\}) \wedge A \neq 1$.

## 4.4 Vincolo di funzione parziale

Il vincolo di funzione parziale nel linguaggio CLP($\mathcal{PF}$) ha la forma $pfun(r)$ dove $r$ è una funzione parziale. Il vincolo è soddisfatto se e solo se $r$ soddisfa le condizioni di funzione della matematica cioè
$\forall x, y, y' : ([x, y] \in r \wedge [x, y'] \in r \rightarrow y = y')$.
Le regole sono le seguenti:

$$\frac{empty(r)}{\{pfun(r)\} \rightarrow \{\}} \tag{4.10}$$

$$\frac{r = \{t_1 \ldots t_n\} \quad n \geq 1 \quad is\_ground(dom(r)) \quad is\_pfun(r)}{\{pfun(r)\} \rightarrow \{\}} \tag{4.11}$$

$$\frac{r = \{[x, y]|rr\} \qquad \neg(is\_ground(dom(r)))}{\begin{array}{c} pfun(r)\} \rightarrow \{dom(rr, d), x \text{ nin } d, [x, y] \text{ nin } rr, pfun(rr)\} \\ o \\ \{rr = \{[x, y]|s\}, dom(s, d), x \text{ nin } d, [x, y] \text{ nin } s, pfun(s)\} \end{array}} \tag{4.12}$$

### 4.4.1 Implementazione in JSetL

In JSetL l'uso del vincolo si traduce nell'invocare il metodo `pfun()` che equivale a $pfun(this)$. Le regole sono state implementate dal metodo `pfunRule()` seguendo la struttura dei precedenti vincoli.

```
private void pfunRule(LMap r,AConstraint s) throws Failure{

  /** CASI BASE **/
  /** Implementazione della regola 4.10**/
 if(r.isInit() && r.isEmpty()){
  s.solved = true;
  return;
 }
```

```
/** Implementazione della regola 4.11**/
if(r.getSize() >= 1 && LMap.isGround(r) && LMap.isPfun(r)){
  s.solved = true;
  return;
}

/** Implementazione della regola 4.12**/
if(r instanceof LMap && r.isInit() ){

  Pair head = (Pair)r.getOne();
  LMap tail = (LMap)r.removeOne();

  switch(s.caseControl){
  /** Primo caso non deterministico **/
  /** La coppia estratta e' unica nella funzione **/
  case 0:
       /** Stato del solver in questo punto dell'esecuzione **/
       VarState statoVarIn = Solver.B.getVarState();
       StoreState statoStoreIn = Solver.B.getStoreState(s);
       /** Creazione del punto in cui tornare nel caso di
         backtracking **/
       Solver.B.addChoicePoint(1, statoVarIn, statoStoreIn);

       LSet dom = new LSet();

       Solver.add(new
           AConstraint(tail,Environment.mapDomCode,dom));

       if(head.isInit()){
           Solver.add(new
               AConstraint(head.getFirst(),Environment.ninCode,dom));
       }else{
           head = new Pair(new LVar(),new LVar());
           Solver.add(new
               AConstraint(head.getFirst(),Environment.ninCode,dom));
       }

       Solver.add(new
           AConstraint(head,Environment.ninCode,tail));


       r = tail;
```

```
            s.arg1 = tail;
            pfun(s);

            Solver.storeInvariato = false;

            return;

          /** Secondo caso non deterministico **/
          /** La coppia estratta e' una ripetizione di un elemento
          della funzione **/
        case 1:
            s.caseControl = 0;

            Solver.add(new
                AConstraint(head,Environment.inCode,tail));

            s.arg1 = tail;

            pfun(s);
            Solver.storeInvariato = false;

            return;
            }
        }

        s.solved = true;
    }
```

Il vincolo pfun è non deterministico, come si può notare dalle regole. In JSetL questo viene realizzato mediante l'uso del metodo `addChoicePoint()` della classe `Backtracking`. Il metodo prende tre parametri, un'intero, lo stato delle variabili non inizializzate e lo stato del constraint store in quel momento. L'intero passato come parametro viene utilizzato per selezionare la regola corretta da applicare. All'interno del comando `switch` sono implementate le regole identificate dai `case`. Come parametro per selezionare il caso corretto viene utilizzato l'attributo `caseControl` dell'`AConstraint` trattato, di default il campo è settato con il valore 0 ma in caso di backtracking viene impostato il valore intero passato come parametro del metodo `addChoicePoint()`.

### 4.4.2 Esempi d'uso

Di seguito verranno mostrati degli esempi di uso del vincolo in alcuni programmi.

**Esempio 1:**

Sia data la funzione $f = \{[1,2],[3,4],[5,6],[7,8]\}$ verificare che sia una funzione parziale.

```
public static void main() throws Failure{

  /**     Creazione del solver    **/
  SolverClass solver = new SolverClass();

  /**     Creazione della funzione f  **/
  LMap f = LMap.empty()
            .ins(new Pair(1,2))
            .ins(new Pair(3,4))
            .ins(new Pair(5,6))
            .ins(new Pair(7,8));

   /**    Verifica delle proprieta' **/
   solver.add(f.pfun());
   solver.solve();
   solver.showStore();

}
```

La funzione usata nel test è il caso particolare in cui ogni elemento di f è ground, cioè definito, quindi il solver per stabilire se rispetta o no le proprietà di funzione parziale si limiterà a invocare il metodo ausiliario `is_pfun()` fornito dalla classe.

**Esempio 2:**

Sia data la funzione $f = \{[1,2],[3,4],?\}$ parzialmente specificata, verificare per quali condizioni è una funzione parziale.

```
public static void pfunProve() throws Failure{

       /**    Creato il solver    **/
```

```
        SolverClass solver = new SolverClass();

        /**   Creata la funzione f con un elemento non definito **/
    LMap f = LMap.empty()
             .ins(new Pair(1,2))
             .ins(new Pair())
             .ins(new Pair(3,4))
             .setName("f");

    f.output();

    int i = 1;

    /**   Risoluzione del vincolo **/
    solver.add(f.pfun());
    solver.solve();

    System.out.println("Soluzione #"+i);
    f.output();

    solver.showStore();

        /**   Ricerca delle successive soluzioni **/
    while(solver.nextSolution()){

             f.setName("f");
             ++i;
             System.out.println("Soluzione #"+i);
             f.output();
             solver.showStore();

    }

  }
```

L'output del programma è il seguente:

*Soluzione* 1
$f = \{(3,4),(\_?,\_?),(1,2)\}$
Store: 3 neq _? AND _? neq 1

*Soluzione* 2
$f = \{(3,4),(1,2)\}$
Store: 3 neq _?

*Soluzione* 3
$f = \{(3,4),(1,2)\}$
Store: _? neq 1

Il solver trova tre soluzioni al problema. La prima soluzione suppone che l'elemento non specificato sia diverso da tutti gli altri elementi della funzione, in particolare il primo elemento della coppia. Le soluzioni successive trovate ipotizzano che l'elemento non specificato sia una ripetizione di una delle altre componenti, per cui viene generata una soluzione per ogni altro elemento della funzione.

## 4.5 Vincolo di composizione funzionale

Il vincolo di composizione funzionale nel linguaggio CLP($\mathcal{PF}$) ha la forma *comp(r,s,q)* dove $r, s$ e $q$ sono funzioni parziali. Il vincolo è soddisfatto se e solo se $q = r \circ s$.
Le regole sono le seguenti:

$$\frac{empty(r)}{\{comp(r,s,q)\} \rightarrow \{q = \emptyset\}} \tag{4.13}$$

$$\frac{empty(s) \qquad \neg empty(r)}{\{comp(r,s,q)\} \rightarrow \{q = \emptyset\}} \tag{4.14}$$

$$\frac{empty(q) \qquad \neg empty(r) \qquad \neg empty(s)}{\{comp(r,s,q)\} \rightarrow \{ran(r,rr), dom(s,ds), disj(rr,ds)\}} \tag{4.15}$$

$$\frac{q = \{[x,z]|rq\} \qquad \neg empty(r) \qquad \neg empty(s)}{\begin{array}{c}\{comp(r,s,q)\} \rightarrow \{r = \{[x,y]|rr\}, s = \{[y,z]|rs\} \\ , [x,z] \text{ nin } rq, [y,z] \text{ nin } rs, comp(rr,s,rq)\}\end{array}} \tag{4.16}$$

$$\frac{q \in \mathcal{V} \quad r = \{[x,y]|rr\} \quad \neg empty(s) \quad s \notin \mathcal{V}}{\begin{array}{c}\{comp(r,s,q)\} \rightarrow \{s = \{[y,z]|rs\}, q = \{[x,z]|rq\}, [x,y] \text{ nin } rr \\ , [y,z] \text{ nin } rs, comp(rr,s,rq)\} \\ o \\ \{dom(s,ds), y \text{ nin } ds, [x,y] \text{ nin } rr, comp(rr,s,q)\}\end{array}} \tag{4.17}$$

### 4.5.1 Implementazione in JSetL

In JSetL l'uso del vincolo si traduce nell'invocare il metodo `comp(LMap s,LMap q)` che equivale a $comp(this, s, q)$. Le regole sono state implementate dal metodo `compRule()` e seguono la stessa struttura dei vincoli precedenti.

Le regole 4.13 ,4.14 rappresentano i casi base in cui una delle due funzioni è vuota per cui nell'implementazione il vincolo $comp(r, s, q)$ viene sostituito con il vincolo di uguaglianza tra q e l'insieme vuoto come mostrato di seguito:

```
if(r.isInit() && r.isEmpty()){
  Solver.add(new
      AConstraint(q,Environment.eqCode,LMap.empty()));
  c.solved=true;
  return;
}

if(s.isInit() && s.isEmpty() && r.isInit() && !r.isEmpty()){
```

```
    Solver.add(new
        AConstraint(q,Environment.eqCode,LMap.empty()));
    c.solved=true;
    return;
  }
```

La regola 4.15 descrive il caso in cui r e s sono funzioni non vuote e il dominio di s è disgiunto dal codominio r. In questa regola vengono usati i vincoli *ran* e *dom* descritti in precedenza.

```
  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.ranCode,rr));
        Solver.add(new AConstraint(s,Environment.mapDomCode,ds));
        Solver.add(new AConstraint(rr,Environment.disjCode,ds));
    c.solved = true;
    return;
  }
```

La regola 4.16 ricostruisce le funzioni r e s, qualora non fossero inizializzate, partendo dalla funzione q che è la composizione delle due.

```
  if(q.isInit() && !q.isEmpty() && (!s.isInit() ||
     !r.isInit())){

    Pair hq = (Pair)q.getOne();
    LMap rq = (LMap)q.removeOne();

    LVar y = new LVar();


    if(!r.isInit()){
      Pair elem_r = new Pair(hq.getFirst(),y);
      Solver.add(new AConstraint(elem_r,Environment.inCode,r));
    }
```

```
    if(!s.isInit()){
      Pair elem_s = new Pair(y,hq.getSecond());
      Solver.add(new AConstraint(elem_s,Environment.inCode,s));
    }

    if(rq.isInit() && !rq.isEmpty()){
      c.arg3 = rq;
      comp(c);
    }
    Solver.storeInvariato = false;
    return;
  }
```

La regola 4.17 esegue la composizione delle funzioni r e s qualora q non fosse inizializzata. La regola è non deterministica e prevede nel primo caso di costruire l'elemento di q a partire dalla coppia $[x, y] \in r$ con $y \in dom(s)$ e nel secondo di aggiungere il vincolo $y \notin dom(s)$.

```
  if(r.isInit() && !q.isInit() && s.isInit() && !s.isEmpty()){

    Pair hr = (Pair)r.getOne();
    LMap rr = (LMap)r.removeOne();
    LVar y = new LVar();
    LSet doms = new LSet();

    Solver.add(new AConstraint(s,Environment.mapDomCode,doms));

    switch (c.caseControl) {
    /** Primo caso non deterministico
        costruzione dell'elemento di q **/
    case 0:
       /** Creazione del punto in cui fare backtracking **/
      VarState statoVarPfun = Solver.B.getVarState();
      StoreState statoStorePfun = Solver.B.getStoreState(c);
      Solver.B.addChoicePoint(1, statoVarPfun, statoStorePfun);

      LVar x = new LVar();
      LVar z = new LVar();

      Pair hr_t = new Pair(x,y);
      Pair hs = new Pair(y,z);
      Pair hq = new Pair(x,z);
```

```
  LMap auxMap = new LMap();
  LMap tmpMap = new LMap();

  LSet domr = new LSet();
  LSet rans = new LSet();

  Solver.add(hs.in(s));

  if(hr.isInit()){
    Solver.add(new
        AConstraint(hr.getSecond(),Environment.eqCode,y));

    hq.setFirst(hr.getFirst());
    Solver.add(new AConstraint(hr,Environment.ninCode,rr));

  }else{
    Solver.add(hr_t.in(r));
  }

  Solver.add(new
      AConstraint(tmpMap,Environment.eqCode,auxMap.ins(hq)));
  Solver.add(new AConstraint(q,Environment.eqCode,tmpMap));

  c.arg1 = rr;

  c.arg3 = auxMap;
  comp(c);
  Solver.storeInvariato = false;
  return;
/** Secondo caso non deterministico
    inserzione del vincolo y nin dom(s)**/
case 1:
  c.caseControl = 0;
  c.arg1 = rr;

  if(hr.isInit()){
    Solver.add(new
        AConstraint(hr.getSecond(),Environment.ninCode,doms));
  }

  Solver.add(new AConstraint(hr,Environment.ninCode,rr));
  comp(c);
```

```
        Solver.storeInvariato = false;
        return;
      }
    }

  }
```

### 4.5.2 Esempi d'uso

Di seguito verranno mostrati degli esempi d'uso del vincolo in alcuni programmi.

**Esempio 1:**

Siano date le funzioni $r = \{[1,2],[3,4]\}$ e $s = \{[2,5],[4,6]\}$ trovare la funzione $q = r \circ s$.

```
public static void main() throws Failure{

  /** Creo il solver **/
  SolverClass solver = new SolverClass();

  /** Definisco le funzioni **/
  LMap r = LMap.empty().ins(new Pair(1,2)).ins(new Pair(3,4));
  LMap s = LMap.empty().ins(new Pair(2,5)).ins(new Pair(4,6));
  LMap q = new LMap();

  /** Aggiungo il vincolo e lo risolvo **/
  solver.add(r.comp(s,q));
  solver.solve();

  q.setName("q");
  q.output();
}
```

Il risultato della computazione è la funzione q voluta come mostrato dall'output:

$q = \{[2,5],[3,6]\}$

**Esempio 2:**

Sia data la funzione $q = \{[1,2],[3,4]\}$ si vogliono ottenere le possibili funzioni a e b tali che $q = a \circ b$.

```
public static void main() throws Failure{

  /** Definisco il solver **/
  SolverClass solver = new SolverClass();

  /** Definisco le funzioni **/
  LMap a = new LMap();
  LMap b = new LMap();
  LMap acompb = LMap.empty().ins(new Pair(1,'a')).ins(new
      Pair(3,6));

  /** Aggiungo il vincolo al solver e lo risolvo **/
  solver.add(a.comp(b, acompb));
  solver.solve();

  a.setName("a");
  b.setName("b");
  a.output();
  b.output();
}
```

Il risultato della computazione è una rappresentazione delle possibili funzioni a e b che rispettano il vincolo:

$a = \{(3, \_?), (1, \_?)\}$
$b = \{(\_?, 2), (\_?, 4)\}$

## 4.6 Vincoli composti

I vincoli composti hanno regole di riscrittura particolari che esprimono il vincolo attraverso un'uguaglianza logica con una congiunzione di vincoli primitivi. Di seguito verrà usato come esempio di questa categoria di vincoli il vincolo dominio ristretto espresso nel linguaggio CLP($\mathcal{PF}$) con la sintassi $dres(dr, r, s)$ dove $r$ e $s$ sono funzioni e $dr$ è il dominio ristretto. Il vincolo $dres$ viene ottenuto con la congiunzione dei seguenti vincoli:

$$dres(dr, r, s) \Longleftrightarrow dom(r, domr) \wedge dom(s, ds) \wedge inters(dr, domr, ds) \wedge subset(s, r)$$

### 4.6.1 Implementazione in JSetL

In JSetL l'uso del vincolo si traduce nell'invocare il metodo `dres(LMap s, LSet a)` che equivale a $dres(a, this, s)$. Il codice del metodo è il seguente:

```
public Constraint dres(LMap s,LSet a){
       /** Dichiarazione delle variabili ausiliarie che
           rappresentano
            il dominio di r e il dominio di s**/
     LSet dr = new LSet();
     LSet ds = new LSet();

     /** Vengono creati separatamente i vincoli coinvolti **/
     Constraint dom1 = new Constraint(this,
        Environment.mapDomCode, dr);
     Constraint dom2 = new Constraint(s, Environment.mapDomCode,
        ds);
     Constraint inters = a.inters(dr, ds);
     Constraint subs = s.subset(this);

     /** I vincoli vengono congiunti e restituiti dal metodo **/
     return dom1.add(dom2).add(inters).add(subs);
   }
```

### 4.6.2 Esempi d'uso

Di seguito verranno mostrati degli esempi d'uso del vincolo in alcuni programmi.

#### Esempio 1:

Sia $r = \{[1, 2], [3, 4], [5, 6]\}$ una funzione e $f = \{[1, 2], [3, 4]\}$ una sua restrizione si vuole trovare il dominio ristretto $dr$ di $r$.

```
public static void main() throws Failure{
    /** Dichiarazione del solver **/
    SolverClass solver = new SolverClass();

    /** Definizione delle funzioni **/
    LMap r = LMap.empty().ins(new Pair(1,2)).ins(new
        Pair(3,4)).ins(new Pair(5,6));
    LMap f = LMap.empty().ins(new Pair(1,2)).ins(new Pair(3,4));

    /** Definizione dell'insieme del dominio ristretto **/
    LSet dr = new LSet();

    /** Aggiunta e risoluzione del vincolo dres**/
    solver.add(r.dres(f,dr));
    solver.solve();
    dr.setName("dr");
    dr.output();
}
```

L'output del codice è il seguente:

$dr = \{1, 2|\_?\}$

# Capitolo 5

# Conclusioni e sviluppi futuri

In questa tesi è stato mostrato come integrare le funzioni parziali all'interno di un linguaggio di programmazione ad oggetti come Java attraverso l'utilizzo della libreria JSetL. La definizione data delle funzioni parziali e la presenza in JSetL degli insiemi logici e dei vincoli ad essi associati ha permesso di integrarle in modo semplice come tipo primitivo della libreria sfruttandone la rappresentazione insiemistica. Alla libreria sono stati aggiunti inoltre anche i vincoli e le classi necessarie per trattare questa struttura dati.

Per quanto riguarda gli sviluppi futuri della struttura delle funzioni parziali nel capitolo 1 viene accennato a come il solver che si ottiene implementando le regole di riscrittura non sia completo. Ciò è dovuto alla presenza di vincoli irriducibili che non sempre si rivelano soddisfacibili. Un esempio è il seguente:

$$dom(X, D1) \wedge dom(X, D2) \wedge D1 \neq D2$$

Il vincolo si dimostra facilmente non soddisfacibile ma il solver non è in grado di provare tale insoddisfacibilità. Una possibile soluzione[2] è data dall'estendere il solver aggiungendo una serie di *regole di inferenza* che trattano vincoli di questo tipo, come la seguente:

$$\frac{}{\{dom(r,a),dom(r,b)\} \rightarrow \{dom(r,a),a=b\}}$$

Il vincolo precedente verrebbe riscritto in $dom(X, D1) \wedge D1 = D2 \wedge D1 \neq D2$ permettendo al solver di rilevare il fallimento. Inoltre anche in presenza di un solver completo queste regole di inferenza sono utili a rendere più efficiente la risoluzione dei vincoli in quanto permettono di rilevare questi particolari tipi di fallimenti prima di espandere i vincoli coinvolti.

Un'altra soluzione è rendere il solver completo modificando le regole di riscrittura secondo le indicazioni date in [3]. Nel documento viene specificato

come aggiungendo le seguenti regole:

$$\frac{r \in \mathcal{V} \qquad a = \{y_1|y\} \qquad \neg empty(a)}{\{ran(r,a)\} \rightarrow \{ran(r_1,\{y_1\}), ran(r_2,y), un(r_1,r_2,r), y_1 \text{ nin } y\}} \tag{5.1}$$

$$\frac{r \in \mathcal{V} \qquad a = \{y\}}{\{ran(r,a)\} \rightarrow \{comp(r,\{[y,y]\},r), r \text{ neq } \emptyset\}} \tag{5.2}$$

al vincolo ran e rimuovendo le disuguaglianze del tipo $X$ neq $t$, dove $X$ è una variabile che occorre come argomento di uno dei vincoli primitivi un, dom, ran o comp, si ottiene un solver completo. Tutto questo è facilmente implementabile in JSetL in modo del tutto analogo a quanto fatto in questo lavoro di tesi modificando solo i metodi opportuni, come `mapRangeRule()` per quanto riguarda le nuove regole per ran, o aggiungendone di nuovi per quanto riguarda le regole di inferenza e le disuguaglianze.

# 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] Maximililiano Cristià, Gianfranco Rossi and Claudia Frydman
*Adding Partial Functions to Constraint Logic Programming with Sets*
Theory and Practice of Logic Programming, vol. 15(4-5), 651-665, 2015.

[3] Maximililiano Cristià and Gianfranco Rossi
*A Complete Solver for Partial Functions in Constraint Logic Programming*

[4] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi
*Sets and Constraint Logic Programming*
ACM Transaction on Programming Language and Systems, Vol. 22 (5), Sept. 2000, 861-931

[5] Maximililiano Cristià, Gianfranco Rossi and Claudia Frydman
*Proofs for Adding Partial Functions to Constraint Logic Programming with Sets*
**http://www.math.unipr.it/ gianfr/SETLOG/setlogpf_proofs.pdf**

[6] Gianfranco Rossi e Roberto Amadini
*JSetL User's Manual*
**http://cmt.math.unipr.it/jsetl/JSetLUserManual-v.2.3.pdf**

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

[8] {log} Home page
**http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html**