UNIVERSITÀ DI PARMA

DIPARTIMENTO DI SCIENZE
MATEMATICHE FISICHE E INFORMATICHE

Corso di Laurea in Informatica

Tesi di Laurea

Estensione della libreria Java JSetL con gli Insiemi Intensionali Ristretti

Relatore:
Prof. Gianfranco Rossi

Candidato:
Andrea Guerra

Anno Accademico 2016/2017

Indice

Capitolo 1

Introduzione

Un insieme è una collezione di elementi dove l'ordine e la ripetizione degli elementi non conta.

Un modo per descrivere gli insiemi è quello di elencarne gli elementi appartenenti. In questo caso parliamo di *insiemi estensionali*, in quanto sono denotati da un'enumerazione esplicita dei propri elementi.

Esempio 1.
L'insieme dei primi 5 numeri pari è denotato dall'insieme $\{0, 2, 4, 6, 8\}$
□

Un altro modo per descrivere un insieme è quello di rappresentarlo attraverso una proprietà.

È questo il caso degli *insiemi intensionali*, ossia una collezione di elementi dove l'appartenenza di un elemento all'insieme è decisa da un *proprietà*.

Esempio 2.
Il seguente *insieme intensionale*: $\{x : [0, 10] \mid x \ mod \ 2 = 0\}$ denota l'insieme dei valori x, compreso nell'intervallo $[0, 10]$, che soddisfa il vincolo $x \ mod \ 2 = 0$. Quindi, tale *insieme intensionale* denota l'*insieme estensionale* $\{0, 4, 6, 8, 10\}$
□

L'obiettivo di questa tesi è quello di estendere la libreria Java `JSetL` con una forma particolare di *insiemi intensionali*, ossia gli *insiemi intensionali* ristretti *(RIS)*. Parliamo di *insiemi intensionali* ***ristretti*** perchè nella loro definizione vengono introdotte varie restrizioni sintattiche che garantiscono, tra l'altro, che i *RIS* possano denotare soltanto collezioni *finite* di elementi.

`JSetL` è una libreria Java, sviluppata con lo scopo di integrare il *paradigma imperativo della programmazione orientata agli oggetti* offerta dal linguaggio Java, con il *paradigma della programmazione logica con vincoli(CLP).*
`JSetL` offre questa possibilità tramite la gestione di: variabili logiche, unificazione, strutture dati ricorsive, risoluzione di vincoli, non-determinismo.
In sostanza, `JSetL` è una libreria che implementa in Java il linguaggio logico a vincoli *CLP($\mathcal{SET}$)*[] permettendo la gestione di insiemi (anche parzialmente specificati) e dei rispettivi vincoli.
I vincoli, nello specifico, riguardano operazioni della teoria base degli insiemi (per esempio appartenenza, unione, intersezione, etc.), ma anche operatori di confronto (uguaglianza, disuguaglianza, etc.) tra interi. `JSetL` implementa un proprio *risolutore di vincoli*, che è di base lo stesso del linguaggio *CLP($\mathcal{SET}$)*, che permette di risolvere vincoli sfruttando *punti di scelta*, *backtracking* e *non determinismo*. Ciò comporta un'elevata complessità computazionale per problemi complessi, ma questo non importa in quanto l'efficienza non è l'obbiettivo primario di `JSetL`. Utilizzando `JSetL` sarà possibile creare programmi che sfruttano le potenzialità e la leggibilità dei *linguaggio dichiarativi* mantenendo, allo stesso tempo, tutte le caratteristiche dei programmi Java convenzionali.

L'estensione che si vuole ottenere permette all'utente di creare e gestire oggetti che rappresentano gli *insiemi intensionali ristretti*, potendo applicare su di essi le operazioni fondamentali come l'uguaglianza, la disuguaglianza, l'appartenenza e la non appartenenza ($=$, $\neq$, $\in$, $\notin$). Quest'estensione favorisce la possibilità di esprimere problemi complessi in maniera più semplice e compatta. Infatti, come vedremo, la notazione *intensionale* con i relativi vincoli associati, è molto espressiva e consente, tra le altre cose, di rappresentare *quantificatori universali ristretti* e *funzioni parziali.*

Capitolo 2

Insiemi Intensionali Ristretti

Questo capitolo si occupa di descrivere, in modo informale, la notazione degli *insiemi intensionali ristretti* e introdurne il funzionamento attraverso degli esempi. Inoltre nella sezione 2.4 verranno introdotte le regole per la riscrittura di vincoli associati ai *RIS*.

2.1 Restricted Intensional Sets (RIS)

Abbiamo detto che gli *insiemi intensionali* si differenziano dagli *insiemi estensionali* in quanto sono insiemi descritti da una proprietà che gli elementi devono soddisfare invece che denotati da un'enumerazione esplicita. Utilizzeremo la notazione:
$\{x : D \mid F\}$
per descrivere l'insieme di elementi x appartenenti a D che soddisfano il vincolo F.

Esempio 1. Sia D un insieme di interi, l'insieme
$\{x : D \mid x > 0 \}$ denota l'insieme di interi positivi appartenenti a D.
□

L'elemento sintattico $\{x : D \mid F\}$ possiede due parti:

- $x : D$, fornisce una collezione di valori D per x;
- F, che è il vincolo che funge da filtro, portando fuori solo i valori che soddisfano F

Un'estensione di tale elemento sintattico è data dall'aggiunta del termine $P(x)$, che rappresenta un'espressione che viene valutata su quei valori di x

che soddisfano il vincolo F.

Esempio 2. Sia D un insieme di interi, l'insieme $\{x : D \mid x > 0 \bullet x * x\}$ denota l'insieme dei quadrati degli interi positivi appartenenti a D.
□

Quindi, si ha che un *insieme intensionale ristretto* ha la forma:
$\{x : D \mid F \bullet P(x)\}$
dove:

- x, chiamato *control Term*, è un termine (in particolare una variabile);
- D, chiamato *domain*, è un insieme (*estensionale o intensionale*) o una variabile;
- F, chiamato *filter*, è un vincolo (ovvero una formula in un opportuno linguaggio logico di prim'ordine);
- $P(x)$, chiamato *pattern*, è un'espressione contenente x;

Un *insieme intensionale*, informalmente, è interpretato come:
l'insieme di tutti gli elementi risultanti dalla valutazione di $P(c)$ per ogni c appartenente all'insieme D per il quale $F(c)$ è soddisfatta.

Più formalmente, se $x_1, x_2, ..., x_n (n > 0)$ sono tutte e sole le variabili che occorrono in c, l'*insieme intensionale* $\{c : D \mid F \bullet P(c)\}$ denota l'insieme:

$$\{y : \exists\ x_1, x_2, ..., x_n\ (c \in D \wedge F \wedge y = P(c))\}$$

È da notare che le variabili $x_1, x_2, ..., x_n$, sono variabili il cui scope è limitato al *RIS* stesso. Inoltre, possono esserci occorrenze di *variabili libere* nel *filter* e nel *pattern* del *RIS*.
Quando $P(x)$ coincide con il *control term* x, allora l'*insieme intensionale ristretto* potrà essere scritto semplicemente come $\{x : D|F\}$.

E' da notare che se D e' un *insieme finito* allora un *RIS* puo' descrivere solo una collezione di elementi *finita*.

Se il dominio di una variabile è ristretto ad un unico valore, allora la variabile è detta essere *inizializzata*. Altrimenti, la variabile è detta essere *non inizializzata*.

Una collezione, dove almeno un elemento o una parte della collezione stessa,

è una variabile logica *non inizializzata* è detta essere una collezione *parzialmente specificata*.

Anche un *RIS* può essere parzialmente specificato. In particolare, siccome il dominio può essere, a sua volta, un insieme *parzialmente specificato* o una variabile, i *RIS* sono una collezione finita ma *illimitata*.

Esempio

- $\{x : \{1, b, 3\} | x > 0 \wedge x < 7\}$ dove b è una variabile *non inizializzata*;
- $\{x : \{1, 2, |Y\} | x > 0 \wedge x < 4\}$ dove Y è una variabile *non inizializzata*;
- $\{x : D | x > 0 \wedge x < 4\}$ dove D è una *non inizializzata*.

□

2.2 I RIS in un linguaggio logico a vincoli

I *RIS* possono essere inseriti all'interno di un *linguaggio logico a vincoli*.
In questo contesto, un **vincolo atomico** e' un predicato che stabilisce una restrizione su un intervallo di valori che una o piu' variabili possono assumere. Ad es. $x > y + 1, x \in D$, ecc...
Un **vincolo (generale)** e' una *congiunzione* o *disgiunzione* di *vincoli atomici* o loro *negazioni*. Ad es, $x > 0 \wedge x < 4$.
Risolvere un *problema con vincoli*, significa associare ad ogni variabile del problema un valore nel rispettivo dominio tale che tutti i vincoli legati al problema siano soddisfatti.
La risoluzione di vincoli è affidata ad un *risolutore di vincoli (solver)*.
Una proposta di utilizzo dei *RIS* all'interno di un *linguaggio logico a vincoli* è quella in [xxx], in cui i *RIS* vengono aggiunti al *linguaggio logico* con insiemi *CLP($\mathcal{SET}$)*.

CLP($\mathcal{SET}$) integra la capacità della classica *programmazione logica* con la capacità computazionale fornita dagli insiemi e dai vincoli sugli insiemi.

In dettaglio, *CLP($\mathcal{SET}$)* consente la gestione di *insiemi estensionali* anche *annidati* e *parzialmente specificati*. È dotato di un proprio *risolutore di vincoli*, chiamato $SAT_{\mathcal{SET}}$, che è dimostrato essere una *procedura decisionale* per risolvere le formule appartenenti al dominio $\mathcal{SET}$.
Una caratteristica importante di *CLP($\mathcal{SET}$)* è la sua capacità di estendere l'*unificazione* in modo che sia capace di trattare le variabili logiche che

rappresentano gli insiemi e gli elementi degli insiemi. *CLP($\mathcal{SET}$)* 'e in grado di trattare anche formule aritmetiche applicate a variabili logiche intere, utilizzando il supporto del risolutore di vincoli di *CLP($\mathcal{FD}$)*.

In [], il *linguaggio logico CLP($\mathcal{SET}$)* viene esteso con nuovi termini insiemistici che rappresentano *RIS* e alcuni suoi vincoli insiemistici sono adattati in modo da poter operare anche su *RIS*. Inoltre, gli elementi di un *RIS*, e cioe' il *control term* il *dominio*, il *filtro* e il *pattern*, sono essi stessi termini e vincoli di *CLP($\mathcal{SET}$)*.

Precisamente, un vincolo su un *RIS* è un predicato atomico della forma:

- $A = B$,
- $A \neq B$,
- $x \in A$,
- $x \notin A$,

dove A e B sono insiemi (sia nella forma *intensionale* che *estensionale*) e x è un termine. Si possono creare formule per i *RIS*, partendo dai suoi vincoli e legandoli tra loro, attraverso la congiunzione e la disgiunzione.

Esempio 1.

- $y \in \{x : [0, 10] | x \geq 0\} \wedge y \notin \{x : [0, -10] | x \leq 0\}$
- $C = A \cap B \wedge \{x \in D | x \in A \wedge x \in C\} = C$

□

In [] il *control Term* c e il *Pattern* P richiedono una forma precisa. Rispettivamente, se x e y sono due variabili, c può essere x oppure una coppia ordinata $(x,\ y)$, mentre P può essere c o $(c,\ t)$ oppure $(t,\ c)$, dove t è un termine.

2.3 Applicazioni dei RIS

2.3.1 Quantificatori universali ristretti

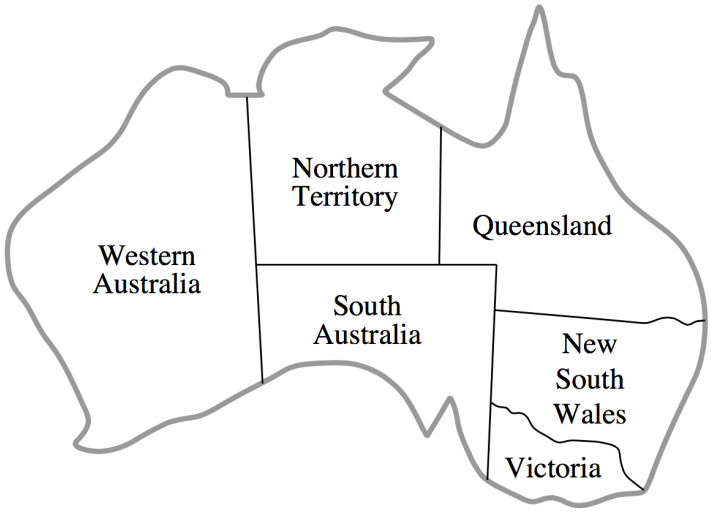
Una caratteristica interessante dei *RIS* è la loro capacità di rappresentare *quantificatori universali ristretti*. Infatti, la formula

$\forall x \in D : F(x)$

può essere facilmente espressa dall'uguaglianza $D = \{x : D | F(x)\}$.

Esempio
La formula $y \in S \wedge S = \{x : S | y \leq x\}$ afferma che y è il minimo di un insieme di interi S. Se, S è il seguente *insieme intensionale* $S = \{2, 4, 1, 6\}$, allora y assume il valore 1. □

Esempio (Colorazione di una mappa)
Data una mappa di n regioni, e un insieme di m colori, trovare un assegnamento di colori alle regioni in modo che per ogni regione, ogni suo vicino abbia un colore differente. Per risolvere questo problema rappresentiamo la mappa come un insieme i cui elementi sono, a loro volta, insiemi che contengono le regioni vicine tra loro. Ogni regione è rappresentata da una variabile logica non instanziata.
Ad esempio la seguente mappa:



È denotata dall'insieme D:
$D = \{ \{wa, nt, sa\}\{nt, q, sa\}\{sa, q, nsw\}\{sa, nsw, v\}\}$ dove $wa, nt, sa...$ sono variabili logiche non inizializzate. I colori sono denotati dall'insieme:
$\{"Red", "Blue", "Green"\}$

Una delle possibili soluzioni a questo problema, utilizzando la notazione *RIS* è la seguente:
$D = \{"Red", "Blue", "Green"\} \wedge D = \{x : D | size(x, 3)\}$
dove, $size(x, 3)$ è il vincolo atomico $|x| = 3$ che fa in modo che la cardinalità dell'insieme x sia il valore 3 (è un modo per dire che ogni variabile appartenente all'insieme di tre variabili assume valori diversi).
□

2.3.2 Funzioni parziali

Un'altra caratteristica importante dei *RIS* è la loro capacità di definire le *funzioni parziali*. In generale, un *RIS* della forma $\{x : D | F \bullet (x, f(x))\}$ dove f è un simbolo di funzione definisce una *funzione parziale* sul dominio D. Tale *RIS* contiene coppie ordinate il cui primo elemento appartiene all'insieme D e, per questo motivo non esistono duplicati. Quindi, se non ci sono due coppie che condividono lo stesso primo elemento, il *RIS* è una funzione. È da notare che la *funzione parziale* è definita specificandone il dominio e l'espressione che la determina.
Attraverso i vincoli sugli insiemi le funzioni possono essere valutate, composte e confrontate. Grazie alla procedura decisionale del risolutore di vincoli, è possibile calcolare direttamente l'inversa di una funzione.

Esempio
Il quadrato di 5 può essere calcolato da:
$(5, y) \in \{x : D \bullet (x, x * x)\}$, il cui risultato è $y = 25$.
Lo stesso *RIS* calcola la radice quadrata di un dato numero:
$(x, 36) \in \{x : D \bullet (x, x * x)\}$, restituisce $x = 6$ e $x = -6$.
□

Esempio
Attraverso la congiunzione di vincoli di appartenenza è possibile rappresentare con i *RIS* funzioni composte. La funzione $f(x) = x^2 + 8$ può essere valutata sul valore 5 come segue:
$(5, y) \in \{x : D \bullet (x, x * x)\} \wedge (y, z) \in \{e : D \bullet (e, e + 8)\}$ restituisce $y = 25$ e $z = 33$. □

2.4 Regole di riscrittura

In [], per poter estendere i vincoli insiemistici ai termini che rappresentano i *RIS* viene illustrato il funzionamento della procedura decisionale $SAT_{\mathcal{RIS}}$. In concreto, esso è un sistema di riscrittura di vincoli.

$SAT_{\mathcal{RIS}}$ funziona in questo modo: data un vincolo (generale) in input, esso viene riscritto, in modo non deterministico, in una nuovo vincolo seguendo le regole di riscrittura interne alla procedura (sono le regole ad essere non-deterministiche), quest'operazione viene ripetuta fin quando non si raggiunge un punto fisso.

Nel caso in cui almeno un vincolo atomico della formula verrà riscritto in *false* allora l'intera procedura restituirà *false*. In caso contrario, la procedura restituisce una collezione di vincoli in *forma risolta*.
Un vincolo è detto in *forma risolta* quando assume una particolare forma, che è dimostrata, essere soddisfacibile. Quando invece la procedura restituisce il valore *false*, vorrà dire che la congiunzione di vincoli iniziale non è soddisfacibile.

Un'altra cosa da notare, è che $SAT_{\mathcal{RIS}}$ riduce in *forma risolta*, solo i vincoli che riguardano gli *insiemi* (*intensionali* ed *estensionali*), mentre i vincoli che coinvolgono termini appartenenti alla teoria $\mathcal{X}$, saranno riscritti dal risolutore di vincoli $SAT_{\mathcal{X}}$, utilizzato internamente alla procedura $SAT_{\mathcal{RIS}}$. La procedura decisionale può trattare uguaglianza, disuguaglianza, appartenenza e non-appartenenza tra insiemi. L'operazione di uguaglianza tra *insiemi estensionali* implementa l'unificazione tra insiemi.

Di seguito sono elencate le regole di riscrittura di vincoli, denotate in questa maniera: $\phi \longrightarrow \Phi_1 \vee ... \vee \Phi_n$, dove ϕ è un vincolo atomico e Φ_i, $i \leq 0$, è una formula di vincoli.
I simboli A, B e D sono *insiemi estensionali*, $\bar{X}$ e $\bar{D}$ sono insiemi non-inizializzati, $\emptyset$ denota l'insieme vuoto, e t, u, y, d e z sono termini appartenenti alla teoria $\mathcal{X}$.

$\bar{X} = A \longrightarrow$ sostituisce $\bar{X}$ con A nel resto della formula $(=_1)$

$\bar{X} = \{t_0, ..., t_n \sqcup \bar{X}\} \longrightarrow \bar{X} = \{t_0, ..., t_n \sqcup N\}$ $(=_2)$

$\{t \sqcup A\} = \{u \sqcup B\} \longrightarrow$ $(=_3)$
$t = u \wedge A = \{u \sqcup B\} \vee$
$t = u \wedge \{u \sqcup A\} = B \vee$
$t = u \wedge A = B \vee$
$A = \{u \sqcup N\} \wedge \{t \sqcup N\} = B$

$\{t \sqcup A\} \neq \{u \sqcup B\} \longrightarrow$ $(=_4)$
$(y \in \{t \sqcup A\} \wedge y \notin \{u \sqcup B\}) \vee$
$(y \notin \{t \sqcup A\} \wedge y \in \{u \sqcup B\})$

$t \in \emptyset \longrightarrow false$ $(\in_1)$

$t \in \{u \sqcup A\} \longrightarrow$ $(\in_2)$
$t = u \vee$
$t \in A$

$t \in \bar{X} \longrightarrow \bar{X} = \{t \sqcup N\}$ $(\in_3)$

$t \notin \emptyset \longrightarrow true$ $(\in_4)$

$t \notin \{u \sqcup A\} \longrightarrow t \neq u \wedge t \notin A$ $(\in_5)$

Fig.1. regole di riscrittura per gli *insiemi estensionali*

$\{x : \emptyset | F \bullet P\} = \emptyset \longrightarrow true$ $(=_5)$

$\{x : \{d \sqcup D\} | F \bullet P\} = \emptyset \longrightarrow \neg F(d) \wedge \{x : D | F \bullet P\} = \emptyset$ $(=_6)$

Se B è un insieme eccetto $\emptyset$:
$\{x : \{d \sqcup D\} | F \bullet P\} = B \longrightarrow$ $(=_7)$
$F(d) \wedge \{P(d) \sqcup \{x : D | F \bullet P\} = B \vee$
$\neg F(d) \wedge \{x : D | F \bullet P\} = B$

$$\{x : \bar{D}|F \bullet P\} = \{y \sqcup A\} \longrightarrow \quad (=_8)$$
$$\bar{D} = \{z \sqcup E\} \wedge F(z) \wedge y = P(z) \wedge \{x : E|F \bullet P\} = A$$

$$\{x : D|F \bullet P\} \neq A \longrightarrow \quad (=_9)$$
$$(y \in \{x : D|F \bullet P\} \wedge y \notin A) \vee$$
$$(y \notin \{x : D|F \bullet P\} \wedge y \in A) \vee$$

$$t \in \{x : \bar{D}|F \bullet P\} \longrightarrow \quad (\in_6)$$
$$d \in \bar{D} \wedge F(d) \wedge t = P(d)$$

$$t \in \{x : \{d \sqcup D\}|F \bullet P\} \longrightarrow \quad (\in_7)$$
$$F(d) \wedge t \in \{P(d) \sqcup \{x : D|F \bullet P\}\} \vee$$
$$\neg F(d) \wedge t \in \{x : D|F \bullet P\}$$

$$t \notin \{x : \{d \sqcup D\} \longrightarrow \quad (\in_8)$$
$$F(d) \wedge t \neq P(d) \wedge t \notin \{x : D|F \bullet P\} \vee$$
$$\neg F(d) \wedge t \notin \{x : D|F|P\}$$

Fig.2. regole di riscrittura per gli *insiemi intensionali*

In **figura 2** sono elencate le regole per risolvere vincoli associati ai *RIS*. Si nota che un *RIS* è trattato come un unico blocco, fino a quando non è necessario identificare almeno uno dei suoi elementi.
Quando ciò avviene, le regole portano il *RIS* nella sua forma estensionale, più precisamente, se y risulta appartenere al *RIS* $\{x : D|F \bullet P\}$ allora il *RIS* viene riscritto nell'*insieme estensionale*: $\{y \sqcup \{x : D'|F \bullet P\}\}$, dove $\{x : D'|F \bullet P\}$ è semanticamente equivalente a $\{x : D|F \bullet P\} \backslash \{y\}$.

Le regole $(=_5)$ - $(=_8)$ consentono l'uguaglianza tra *RIS* ed *insiemi estensionali*.
In particolare, la regola $(=_6)$ rappresenta il *quantificatore universale:*
$\forall x \in D : \neg F(x)$
forzando ogni elemento appartenente al dominio del *RIS* a soddisfare il vincolo ottenuto con la negazione del *filtro*. La regola $(=_8)$ permette l'uguaglianza tra un *RIS* il cui dominio non è inizializzato e un *insieme estensionale*. In queste regole i *RIS* considerati hanno ulteriori restrizioni: 1) il dominio dato non è un altro *RIS*, 2) il *control term* dei *RIS* è la *variabile* x. I casi non considerati (come l'uguaglianza tra una variabile e un *RIS* con dominio non inizializzato) in [] sono dimostrati essere *irriducibili*.

Capitolo 3

Libreria JSetL

Nel capitolo 1, abbiamo iniziato a parlare della libreria Java `JSetL`.
Questo capitolo si occupa di descrivere più in dettaglio `JSetL` presentando le sue principali caratteristiche e la gestione delle sue più importanti strutture dati.

3.1 Variabili logiche

`JSetL` supporta la nozione di variabile logica così come è solitamente vista nei linguaggi di programmazione logica. Le variabilil logiche possono essere sia **inizializzata** che **non-inizializzate**. Il valore di una variabile logica in `JSetL` può essere di qualsiasi tipo.

Definition 3.1.1. Una variabile logica, in `JSetL` è un'istanza della classe `LVar`, creata dall'istruzione:

`LVar` VarName = **new** `LVar`(ExtVarName, VarValue);

dove VarName è il nome della variabile, ExtVarName è un nome esterno opzionale, VarValue è un valore opzionale per la variabile.

Il nome esterno è una stringa che può essere utile quando si vuole visualizzare in output la variabile e i possibili vincoli che la coinvolgono (se omesso, il nome di default è "_?"). Il valore associato ad una variabile logica, chiamato `Lvar-value`, può essere di un qualunque tipo. Esso è incapsulato nella classe `LVar` e, perciò, non può essere acceduto o modificato direttamente ma solo attraverso i metodi della classe.

Definition 3.1.2. Una variabile logica che non ha un `Lvar-value` associato è detta *non-inizializzata* (o un *incognita*). Se una variabile logica ha come `Lvar-value` un oggetto *non inizalizzato* allora anch'essa è detta essere *non inizializzata*. Altrimenti, la variabile logica è *inizializzata*.

Un valore per una variabile logica può essere specificato sia quando la variabile è creata sia come risultato della valutazione di vincoli che coinvolgono la variabile stessa, in particolare, vincoli di uguaglianza. Oggetti `LVar` sono manipolati principalmente attraverso vincoli. I vincoli possono essere usati per impostare il valore di una variabile logica non inizializzata o per controllarla. Nessun vincolo è autorizzato a modificare il valore di una variabile logica sostituendolo con uno nuovo.
I seguenti metodi interni alla classe `LVar`:

```
Constraint eq(Object z)

Constraint neq(Object z)

Constraint in(Set<?> z)

Constraint in(LSet z)

Constraint nin(Set<?> z)

Constraint nin(LSet z)
```

generano i vincoli che rappresentano, rispettivamente, l'uguaglianza (**eq**) che implementa l'unificazione tra la variabile logica `this` e l'oggetto `z`, la disuguaglianza (**neq**), l'appartenenza (**in**) e la non appartenenza (**nin**) ad insiemi logici (o insiemi generici Java). Per finire, la classe `LVar` fornisce un insieme di metodi che permettono la lettura o scrittura del valore di una variabile logica, o di sapere se la variabile è inizializzata o meno e così via.

3.2 Variabili logiche intere

Le *variabili logiche intere* sono un caso particolare di *variabili logiche* dove i valori che può assumere una variabile sono ristretti ad essere numeri interi. In più, una *variabile logica intera* possiede un dominio finito e un *vincolo aritmetico intero* associato (può essere anche vuoto). Il dominio è rappresentato come un *multi-intervallo*, che è l'unione di n ($n > 0$) *intervalli* disgiunti. Un dominio per una *variabile logica intera* v può essere specificato quando v viene creata e aggiornato quando i vincoli che coinvolgono v vengono risolti in maniera tale che venga preservata la consistenza dei vincoli. Quando il dominio di una variabile è ristretto ad un singolo valore, allora la variabile è detta *inizializzata*. Se la risoluzione di un vincolo dovesse portare la variabile logica intera x ad avere un dominio vuoto allora vuol dire che il vincolo che coinvolge x è insoddisfacibile.

I vincoli aritmetici associati con *variabili logiche intere* sono generati dalla valutazione di *espressioni logiche intere*. Un *espressione logica intera* è un insieme di costanti intere e *variabili logiche intere* legate tra loro da operatori matematici. Tali espressioni sono costruite utilizzando gli usuali operatori aritmetici `sum, mul, div, mod, sub`.

La valutazione di un'*espressione logica intera* **e**, produce una nuova *variabile logica intera* X_1 con un vincolo associato della seguente forma:

$$X_1 = e_1 \wedge X_2 = e_2 \wedge ... \wedge X_n = e_n$$

dove $e_1, e_2, ...e_n$ sono sottoespressioni che occorrono in **e** mentre $X_1, X_2, .., X_n$ sono *variabili logiche intere interne* che occorrono in $e_1, e_2, ...e_n$.

Ad esempio, se **e** è l'*espressione logica* `x.sum(y.sub(1))`, dove x e y sono *variabili logiche intere*, la valutazione di **e** restituisce la variabile logica intera X_1 con il seguente vincolo aritmetico associato: $X_1 = x + X_2 \wedge X_2 = y - 1$.

Definition 3.2.1. Una *variabile logica intera*, in `JSetL`, è un'istanza della classe `IntLVar` che estende la classe `LVar`, creata dalla dichiarazione:

`IntLVar` VarName = **new** `IntLVar`(ExtVarName, DomainValues)

dove VarName è il nome della variabile logica intera, ExtVarName è il nome esterno opzionale della variabile e DomainValues è una parte opzionale usata per specificare il dominio della variabile che può essere un multi intervallo di interi, o un singolo intero nel caso in cui la variabile sia inizializzata.

Siccome la classe `IntLVar` estende la classe `LVar`, essa eredita tutti i suoi metodi che sono adattati ad oggetti `IntLVar` o `Integer`. Il valore associato ad una variabile logica intera dev'essere un intero. In particolare, il valore associato ad una variabile logica intera può essere solamente un'istanza di tipo `Integer` o `IntLVar`, o il tipo primitivo `int`. Come un oggetto `LVar`, un oggetto `IntLVar`, può essere *non inizializzato*, questo avviene quando nessun intero è stato associato all'oggetto, oppure è stato associato un altro oggetto *IntLVar non inizializzato*. È possibile costruire una variabile logica intera di `JSetL` specificandone il dominio (utilizzando un oggetto `MultiInterval`, o due interi che denotano un intervallo). Quando questo è omesso assumerà il valore di default [`INF, SUP`] dove `INF` e `SUP` rappresentano, rispettivamente, il più piccolo e il più grande intero rappresentabili all'interno di un multi intervallo.

Oggetti di tipo `IntLVar` possono essere creati anche con i metodi che rappresentano le operazioni aritmetiche `sum, sub, mod, div, mul`. Tali metodi sono invocati su oggetti `IntLVar` e restituiscono oggetti dello stesso tipo, in modo da essere concatenati per creare *espressioni logiche intere*. Una variabile logica intera creata attraverso questi metodi ha associato un vincolo

aritmetico generato dalla valutazione dell'espressione logica intera denotata. Ad esempio, sia $\oplus$ un operatore binario rappresentato da un metodo aritmetico della classe `IntLVar`, l'invocazione del metodo crea una nuova *variabile logica intera* X_1 con un vincolo associato della forma: $X_1 = X_0 \oplus v \wedge C_v \wedge C_0$, dove X_0 è la variabile logica intera rappresentata dall'oggetto d'invocazione, C_0 è il vincolo associato alla variabile X_0, v è l'altro operando dell'operatore binario (può essere un `Integer` o un `IntLVar`) e C_v è il vincolo eventualmente associato a v (se v è di tipo `IntLVar`).
I metodi aritmetici più importanti di `IntLVar` sono:

```
IntLVar sum(Integer k)

IntLVar sum(IntLVar k)

IntLVar sub(Integer k)

IntLVar sub(IntLVar k)

IntLVar mul(Integer k)

IntLVar mul(IntLVar k)

IntLVar div(Integer k)

IntLVar div(IntLVar k)

IntLVar mod(Integer k)

IntLVar mod(IntLVar k)
```

La classe `IntLVar` fornisce i metodi per generare i tradizionali vincoli aritmetici di confronto. Essi sono:

```
Constraint eq(Integer k)

Constraint eq(IntLVar k)

Constraint neq(Integer k)

Constraint neq(IntLVar k)

Constraint le(Integer k)

Constraint le(IntLVar k)

Constraint lt(Integer k)
```

```
Constraint lt(IntLVar k)

Constraint ge(Integer k)

Constraint ge(IntLVar k)

Constraint gt(Integer k)

Constraint gt(IntLVar k)
```

che generano, rispettivamente, i seguenti vincoli: $=, \neq, \leq, <, \geq, >$. Inoltre, `IntLVar` fornisce ulteriori metodi per generare anche altri tipi di vincoli, come vincoli di appartenenza, vincoli di dominio, vincoli di labeling.

3.3 Liste logiche

Una lista logica o semplicemente una lista, è uno speciale tipo di variabile logica il cui valore è una coppia $<$ *elems, rest* $>$, dove l'elemento *elems* (i valori della lista) è una lista $[e_0, ..., e_n]$ con $n \leq 0$, di oggetti di tipo arbitrario e *rest* è una lista non inizializzata o una lista vuota che rappresenta la parte rimanente di l. Quando rest è una lista non inizializzata r, diremo che l rappresenta una lista aperta e utilizziamo la notazione astratta $[e_0, ..., e_n|r]$ per denotarla; invece quando rest è la lista vuota diciamo che l rappresenta una lista chiusa e usiamo la notazione $[e_0, ..., e_n]$ per denotarla. Quando elems è la lista vuota e rest è null, l è la lista vuota e useremo [ ] per denotarla.

Una lista che contiene oggetti logici non inizializzati rappresenta una lista parzialmente specificata. Intuitivamente, alcuni dei suoi elementi sono incognite. Anche una lista aperta ha una parte incognita, e quindi rappresenta anche lei una lista parzialmente specificata.

In `JSetL`, una lista logica è definita come un'istanza della classe `LList`, che estende la classe `LCollection`. In particolare, i valori di una lista (i.e., la parte elems della lista logica) sono istanze della classe `ArrayList` che implementa l'interfaccia `java.util.List`. La classe `LList` fornisce metodi per creare nuove liste, con la possibilità di costruirle a partire da una già esistente, di trattare liste logiche come variabili logiche e di maneggiarle quando i valori della lista sono inizializzati. Come le variabili logiche, alle liste possono essere associati vincoli che implementano le operazioni base su di esse.

Definition 3.3.1. Una lista logica, in `JSetL`, è un'istanza della classe `LList`, creata con la dichiarazione:

`LList` LstName = **new** `LList`(ExtLListName, LListElem)

dove LstName è il nome della lista, ExtLListName è un nome esterno opzionale, e LListElemValues è una parte opzionale utilizzata per specificare gli elementi di una lista e può essere un array di elementi $c_1, ..., c_n$ di un qualunque tipo, o i limiti l e u di un intervallo $[l, u]$ di numeri interi, o un altra lista. La *lista vuota* è denotata con la costante `LList.empty`.

Definition 3.3.2. Gli elementi di una lista che sono loro stessi liste sono detti essere liste *annidate*. Le liste possono essere *annidate* a qualunque profondità

3.4 Insiemi logici

Un insieme logico s (o, semplicemente un insieme s) è uno speciale tipo di variabile logica il cui valore è una coppia <elems, rest >, dove *elems* è un insieme $\{e_0, ..., e_n\}$, con $n \geq 0$, di oggetti di tipo arbitrario, e *rest* è un insieme non inizializzato o un insieme vuoto che indica la parte restante di s. Quando *rest* è un insieme non inizializzato r, diremo che s rappresenta un insieme aperto e utilizzeremo la notazione $\{e_0, ..., e_n|r\}$ per denotarlo; invece, quando *rest* è l'insieme vuoto, noi diremo che s rappresenta un insieme chiuso e useremo la notazione $\{e_0, ..., e_n\}$. Quando elems è l'insieme vuoto e rest è null, s è l'insieme vuoto, denotato in questo modo: {}. Gli insiemi logici sono simili alle liste logiche sotto vari aspetti. In particolare, come le liste logiche, gli insiemi possono rappresentare collezioni parzialmente specificate. La principale differenza con le liste è che in un insieme l'ordine e la ripetizione degli elementi non importa, mentre loro sono importanti nelle liste logiche. Un'altra caratteristica importante, è che la cardinalità di un'insieme parzialmente specificato non è unica (anche se l'insieme è chiuso). Ad esempio, la cardinalità dell'insieme $\{1, x\}$, dove x è una variabile non inizializzata, può essere 1 oppure 2. Essa dipende, rispettivamente, dal fatto che x può essere inizializzato con il valore 1 oppure con un valore diverso.

Definition 3.4.1. In `JSetL` un insieme è un'istanza della classe `LSet`, creata con la dichiarazione:

`LSet` SetName = **new** `LSet`(ExtLSetName, LSetElemValues);

dove SetName, ExtLSetName, LSetElemValues hanno lo stesso significato che hanno nelle liste.

La classe `LSet` estende la classe `LCollection`. I valori dell'insieme (i.e., la parte elems dell'insieme logico) sono istanze della classe `HashSet` che implementa l'interfaccia `java.util.Set`. I metodi forniti dalla classe `LSet` sono gli stessi della classe `LList` ma applicati agli oggetti `LSet`. Fornisce metodi per:

- creare nuovi insiemi logici, anche con la possibilità di creare un nuovo insieme da uno già esistente,

- trattare gli insiemi logici come variabili logiche,

- manipolare gli insiemi con metodi di utilità generale quando gli insiemi sono inizializzati.

Definition 3.4.2. Come nelle liste, gli elementi di un insieme che sono a loro volta oggetti `LSet`, sono detti insiemi *annidati*.

Con gli insiemi logici possiamo generare vincoli che implementano le operazioni base degli insiemi ($=$, $\neq$, $\subseteq, \cup, \cap$, etc.). In dettaglio, siano s_1, s_2, s_3 tre espressioni di tipo `LSet`, i vincoli che si possono generare sono:

- **uguaglianza** e **non uguaglianza** : s_1`.eq(`s_2`)`
- **disgiunzione** e **non disgiunzione** : s_1`.disj(`s_2`),` s_1`.ndisj(`s_2`)`
- **unione** e **non unione** : s_1`.union(`s_2`,` s_3`),` s_1`.nunion(`s_2`,` s_3`)`
- **inclusione** : s_1`.subset(`s_2`)`
- **intersezione** e **non intersezione** : s_1`.inters(`s_2`),` s_1`.ninters(`s_2`)`

3.5 Vincoli

`JSetL`, come abbiamo visto, permette l'utilizzo di vincoli per specificare delle condizioni sulle variabili logiche e sugli insiemi. I vincoli sono gestiti da un *risolutore di vincoli* (un istanza della classe `SolverClass`), che sostanzialmente implementa la procedura decisionale di *CLP(SET)*.

Definition 3.5.1. (Vincoli atomici). Un *vincolo atomico* in `JSetL` è una relazione binaria o ternaria, definita mediante le espressioni:

- `op(e1, e2);`
- `op(e1, e2, e3);`

dove `op` è uno dei metodi di una collezione (`eq, neq, nin, in, subset, union, inters, disj, differ, nunion, ninters, etc.`) e `e1, e2 , e3` sono espressioni il cui tipo dipende da `op`.

Definition 3.5.2. (Vincoli). Un *vincolo* (generale) può essere sia un vincolo atomico che, ricorsivamente, un'espressione della forma:

- v_1.and(v_2)....and(v_n)

dove $v_1, v_2, ..., v_n$ sono vincoli atomici. Ovvero, la congiunzione di due o più vincoli è anch'essa un vincolo.

Un vincolo, in `JSetL`, è un'istanza della classe `Constraint`.
Un *constraint store* di un risolutore di vincoli S contiene la collezione di tutti i vincoli attivi per S nel programma in esecuzione. In `JSetL`, un *constraint store* è un'istanza della classe `Store`. Ogni istanza della classe `SolverClass` possiede un proprio *constraint store*. Per aggiungere un vincolo al *constraint store* di un oggetto `SolverClass` basta invocare il metodo `add` della classe `SolverClass`. La chiamata del metodo avviene in questo modo:

```
S.add(C)
```

che aggiunge il vincolo C al *constraint store* del risolutore di vincoli S. La collezione di vincoli memorizzati in un *constraint store* è interpretata come una congiunzone di vincoli. Una volta che uno o più vincoli sono stati aggiunti nel *constraint store* di un risolutore di vincoli S, è possibile richiedere la risoluzione di tali vincoli con il metodo `solve`:

S.solve()

Tale metodo ricerca, in modo non deterministico, una soluzione che soddisfi tutti i vincoli contenuti nel *constraint store*. Il metodo `solve` effettua la riduzione di una qualunque congiunzione di vincoli atomici in una forma semplificata, chiamata `solved form`, che è dimostrato essere soddisfacibile. Il successo di questo processo di riduzione permette di concludere che il constraint store, nella sua forma originale, è soddisfacibile. Viceversa un fallimento implica l'insoddisfacibilità dei vincoli presenti nel constraint store. In `JSetL`, la risoluzione di un *constraint store* che ha vincoli insoddisfacibili comporta il lancio dell'eccezione `Failure`.

Capitolo 4

Insiemi Intensionali Ristretti in JSetL

4.1 Classe Ris

In `JSetL`, un *insieme intensionale ristretto* è un'istanza della classe `Ris`, che estende la classe `LSet`.
La classe `Ris` offre, oltre ai metodi ereditati, i costruttori per creare gli *insiemi intensionali ristretti*, un metodo per passare alla loro forma estensionale (quando è possibile) e dei metodi per associare vincoli, come uguaglianza e disuguaglianza.

4.1.1 Costruttori

Un *insieme intensionale ristretto* $\{ct : D|F \bullet P\}$ per essere costruito necessita di 4 attributi:

- una *variabile* che rappresenta il *controlTerm* ct;
- un *insieme logico* che rappresenta il dominio di appartenenza del *controlTerm*, ossia il suo *domain* D;
- un vincolo di tipo `Constraint`, che rappresenta il *filter* dell'*insieme intensionale ristretto* F;
- una *variabile logica* per il *pattern* P.

Un oggetto `Ris` è inizializzato se e solo se il suo campo *domain* è anch'esso inizializzato. Inoltre, è da notare che il *filter* dell'*insieme intensionale ristretto* è strettamente dipendente dal tipo del *controlTerm*. Così come il *pattern* in quanto sono entrambi in funzione di esso.

È possibile omettere il *pattern* quando esso corrisponde al *controlTerm*.

`public Ris(IntLVar ct, LSet D, Constraint f, LVar p)`

`public Ris(String n, IntLVar ct, LSet D, Constraint f, LVar p)`
crea un nuovo *insieme intensionale ristretto* così formato:

$\{ct : D \mid F \bullet P\}$.

Se l'argomento `n` è omesso il nome dell'oggetto `Ris` creato sarà automaticamente "?", altrimenti assumerà il valore della stringa `n`.

`public Ris(IntLVar ct, LSet D, Constraint f)`

`public Ris(String n, IntLVar ct, LSet D, Constraint f)`
crea un nuovo *insieme intensionale ristretto* così formato:
$\{ct : D \mid F \bullet ct\}$.

Se l'argomento `n` è omesso il nome dell'oggetto `Ris` creato sarà automaticamente "?", altrimenti assumerà il valore della stringa `n`.

`public Ris(IntLVar ct, LSet D, Constraint f, LList p)`
crea un nuovo *insieme intensionale ristretto*, $\{x : D \mid F \bullet [y, z]\}$. È da notare che l'unica forma che la lista `p` può assumere è quella di una coppia ordinata di variabili logiche. Nel caso la lista sia diversa da questa forma, il metodo non lancia eccezione ma il comportamento del `Ris` non sarà quello che ci aspettiamo.

`public Ris(LSet ct, LSet D, Constraint f)`
crea un nuovo *insieme intensionale ristretto*, $\{ct : D \mid F \bullet ct\}$ dove il filtro può essere un vincolo che coinvolge l'insieme estensionale *ct*. In particolare, per il momento i filtri supportati sono il vincolo `x.size(N)`, `x.union(S2, S3)`, `x.subset(S1)`, ma anche l'uguaglianza e disuguaglianza. È da notare che non è permesso scegliere la forma dell'espressione che denota il pattern, essa sarà semplicemente l'espressione denotata da *ct*.

`public Ris(LList ct, LSet D, Constraint f)`
crea un nuovo *insieme intensionale ristretto*, $\{ct : D \mid F \bullet ct\}$ dove il filtro può essere un vincolo che coinvolge la lista logica *ct*. In particolare, per il momento i filtri supportati sono i vincoli di uguaglianza e disuguaglianza tra liste logiche. È da notare che non è permesso scegliere la forma dell'espressione che denota il pattern, essa sarà semplicemente l'espressione denotata da *ct*.

Esempio 1

```
IntLVar x = new IntLVar("x");
LSet domain = new LSet(new Interval(-10, 10).toSet());
Constraint f = x.mod(2).eq(0);
Ris ris = new Ris(x, domain, f);
```

Crea l'*insieme intensionale ristretto*: $\{x : [-10, 10] | \ x \ mod \ 2 = 0\}$, ovvero, l'insieme di tutti i numeri pari compresi nell'intervallo [-10, 10].

Esempio 2

```
IntLVar x = new IntLVar("x");
Constraint f = x.le(10).and(x.ge(5));
LSet d = new LSet("D");
IntLVar p = x.mul(x);
Ris ris = new Ris(x, d, f, p);
```

Crea l'*insieme intensionale ristretto* non inizializzato
$\{x : D | x \geq 5 \wedge x \leq 10 \bullet x * x\}$,
che denota l'insieme dei quadrati degli interi appartenenti all'insieme logico `D` compresi tra 5 e 10.

Esempio 3

```
IntLVar x = new IntLVar("x");
Constraint f = x.le(10).and(x.ge(5));
LSet d = new LSet("D");
IntLVar z = x.mul(x);
LList p = LList.empty().ins(z).ins(x);
Ris ris = new Ris(x, d, f, p);
```

Crea l'*insieme intensionale ristretto* non inizializzato
$\{x : D | x \geq 5 \wedge x \leq 10 \bullet [x, x * x]\}$,
che denota l'insieme delle coppie $[x, x * x]$ di interi in cui il primo elemento x appartiene all'insieme logico `D` ed è compreso tra 5 e 10.

Esempio 4

```
LSet x = new LSet("x");
Constraint f = x.size(2);
LSet d = new LSet("D");
Ris ris = new Ris(x, d, f, p);
```

Crea l'*insieme intensionale ristretto* non inizializzato $\{x : D | x.size(2) \wedge x \leq 10 \bullet x\}$, che denota l'insieme degli insiemi appartenenti a `D` la cui cardinalità avrà valore 2.

Esempio 4

```
LList x = new LList("x");
Constraint f = x.neq(LList.empty().ins(1).ins(2);
LSet d = new LSet("D");
Ris ris = new Ris(x, d, f, p);
```

Crea l'*insieme intensionale ristretto* non inizializzato $\{x : D | x.neq([2,1]) \wedge x \leq 10 \bullet x\}$, che denota l'insieme delle liste logiche appartenenti a `D` che sono diverse dalla lista logica `[2, 1]`.

4.1.2 Metodi di utilità generale

`public equals(Object o)`

restituisce `true` se quest'*insieme intensionale ristretto* è uguale all'oggetto `o`. In particolare, restituisce `true` quando:

- `o` è un riferimento allo stesso *RIS*;
- `o` è un'istanza della classe `LSet` e non possiede *variaibli logiche non inizializzate* e anche il *domain* dell'oggetto `this` non ne possiede, in questo caso restituisce `true` se la notazione estensionale di `o` e la notazione intensionale dell'oggetto di invocazione rappresentano lo stesso *insieme logico*, nel caso in cui `o` possiede variabili logiche non inizializzate (o il domain dell'oggetto `this`), restituisce `false`;
- `o` è un'istanza della classe `Ris`, e i campi posseduti da entrambi gli insiemi sono uguali.

In tutti gli altri casi, il metodo restituisce `false`.

Esempio 3

```
IntLVar x = new IntLVar("x");
Constraint f = x.mod(6).eq(0);
LSet d = new LSet(new Interval(1, 30).toSet());
Ris ris = new Ris(x, d, f);
// {x : [1, 30] | x mod 6 = 0 @ x}
LSet s = LSet.empty().ins(30).ins(6).ins(12).ins(18).ins(24);
//{24, 18, 12, 6, 30}
ris.equals(s); //true
```

`public LSet espandi()`

restituisce l'*insieme logico estensionale* equivalente a quest'*insieme intensionale ristretto*. Ciò è possibile se e solo se il campo `domain` di questo *insieme logico* non possiede *variabili logiche* non inizializzate. Nel caso in cui il *domain* dell'oggetto `this` possiede variabili logiche non inizializzate viene lanciata l'eccezione `NotInitVarException`.

Esempio 4

```
IntLVar x = new IntLVar("x");
Constraint f =
    x.le(-15).and(x.ge(-25)).or(x.le(25).and(x.ge(15)));
LSet d = new LSet(new Interval(-50, 50).toSet());
IntLVar p = x.mul(x);
Ris ris = new Ris(x, d, f, p);
//{x : D | (x<=-15 && x>=-25) || (x<=25 && x>=15) @ x*x }
ris.espandi();
//?={625,576,529,484,441,400,361,324,289,256,225}
```

`public boolean isEmpty()`

restituisce `true` se il campo `domain` dell'*insieme intensionale logico* è vuoto.

`public boolean isBound()`

restituisce `true` se il campo *domain* dell'*insieme intensionale ristretto* è inizializzato, `false` altrimenti.

`public boolean isGround()`
: restituisce `true` se il campo `domain` dell'*insieme intensionale ristretto* è anch'esso *Ground*.

`public boolean isClosed()`
: restituisce `true` se il campo `domain` dell'*insieme intensionale ristretto* è anch'esso Closed.

`public Constraint getFilter()`
: restituisce il campo `filter` dell'*insieme intensionale ristretto*.

`public LVar getControlTerm()`
: restituisce il campo `controlTerm` di questo *insieme intensionale ristretto*.

`public LVar getPattern()`
: restituisce il campo `pattern` di questo *insieme intensionale ristretto*.

`public String getName()`
: restituisce il nome di questo *insieme intensionale ristretto*.

`public void setName(String n)`
: imposta il nome esterno di questo *insieme intensionale ristretto* con la stringa n.

`public String toString()`
: restituisce la stringa che denota questo *insieme intensionale ristretto* con il seguente formato: $\{x : D \mid F \bullet P\}$

`public void output()`
: stampa il nome di questo *insieme intensionale ristretto* seguito dal simbolo "=" e dalla stringa che rappresenta quest'*insieme intensionale ristretto*

4.1.3 Metodi per generare vincoli

È possibile associare vincoli ad oggetti della classe `Ris`. Per il momento sono stati implementati i vincoli di uguaglianza e disuguaglianza, mentre gli altri vincoli, ereditati dalla classe `LSet`, non sono stati ancora estesi ai *RIS*. Per questo motivo, l'invocazione di essi comporta il lancio dell'eccezione `NotDefConstraintException()`.

`public Constraint eq(LSet s)`
: restituisce il vincolo atomico `this = s`, che unifica questo *insieme intensionale ristretto* con l'*insieme estensionale* `s`;

`public Constraint neq(LSet s)`
: restituisce il vincolo atomico `this` $\neq$ `s`, che richiede che questo *insieme intensionale ristretto* sia diverso dall'*insieme estensionale* `s`.

Siccome `Ris` è una sottoclasse di `LSet` il parametro `s` può essere anche un oggetto `Ris`. E' quindi possibile stabilire vincoli `eq` e `neq` anche tra due oggetti `Ris`.

Esempio 5

```
IntLVar x = new IntLVar("x");
LSet set = new LSet(new Interval(0, 10).toSet());
Constraint f = x.mod(2).eq(0);
IntLVar p = x.mul(2);
Ris ris = new Ris(x, set, f, p);
// {x : [0, 10] | x mod 2 = 0 @ x*2}}
IntLVar y = new IntLVar("y");
LSet set =
   LSet.empty.ins(20).ins(16).ins(12).ins(8).ins(4).ins(0);
// {0, 4, 8, y, 16, 20 }
SolverClass solver = new SolverClass();
solver.add(ris.eq(set));
// {x : [0, 10] | x mod 2 = 0 @ x*2}} = {0, 4, 8, 12, 16, 20}
solver.solve();
```

Esempio 6

```
IntLVar x = new IntLVar("x");
IntLVar d = new IntLVar("d");
LSet set = LSet.empty().ins(d).ins(4).ins(8);
Constraint f = x.ge(0);
IntLVar p = x.mul(2);
Ris ris = new Ris(x, set, f, p);
// {x : {8,4,d} | x mod 2 = 0 @ x*2}}
IntLVar y = new IntLVar("y");
LSet set1 = new LSet(new Interval(-5, 5).toSet);
Constraint f1 = x.mod(2).eq(0);
```

```
Ris ris1 = new Ris(y, set1, f1);
// {y : [-5, 5] | x mod 2 = 0 @ x }
SolverClass solver = new SolverClass();
solver.add(ris.neq(ris1));
//{x : {8,4,d} | x mod 2 = 0 @ x*2}} != {y : [-5, 5] | x mod 2
    = 0 @ x }
solver.solve();
```

4.2 Esempi

In `JSetL`, sono state introdotte le regole per la gestione di vincoli di appartenenza e di non appartenenza ad insiemi di tipo `Ris`. Sfruttando il fatto che `Ris` è una sottoclasse di `LSet`, i metodi `in(LSet s)` e `nin(LSet s)` delle variabili logiche gestiscono in modo diretto i casi in cui il parametro `s` sia un'istanza della classe `Ris`. Di seguito sono riportati alcuni esempi di utilizzo di vincoli dei `Ris`.

4.2.1 Quantificatori universali ristretti con oggetti Ris

La capacità di risolvere i vincoli associati ad oggetti *Ris* il cui dominio può contenere variabili logiche non inizializzate porta a numerosi vantaggi. È interessante vedere come i *quantificatori universali ristretti* possono essere facilmente implementati in `JSetL` sfruttando oggetti `Ris`.

Esempio 6 Il seguente esempio mostra come può essere risolto il problema di trovare il minimo di un insieme di interi {8, 9, 7, 4} (come visto nel capitolo 2).

```
IntLVar y = new IntLVar("y");
IntLVar x = new IntLVar("x");
LSet S = LSet.empty().ins(4).ins(7).ins(9).ins(8);
Constraint f = x.ge(y);
Ris ris = new Ris(x, S, f);
SolverClass solver = new SolverClass();
solver.add(y.in(S));
solver.add(ris.eq(S));
// y in S AND S = {x : S | x >= y @ x}
solver.solve();
```

Esempio 6 Il seguente esempio mostra come può essere risolto il problema della colorazione di una mappa (visto nel capitolo 2).

```
LVar r1 = new LVar("r1");
LVar r2 = new LVar("r2");
LVar r3 = new LVar("r3");
LVar r4 = new LVar("r4");
LVar r5 = new LVar("r5");
LVar r6 = new LVar("r6");

LVar[] regionsArray = {r1,r2,r3,r4,r5,r6};
LSet regions = LSet.empty().insAll(regionsArray);
LSet[] mapArray =
    {LSet.empty().ins(r1).ins(r2).ins(r3),LSet.empty().ins(r2).ins(r3).ins(r4),
    LSet.empty().ins(r3).ins(r4).ins(r5),
    LSet.empty().ins(r3).ins(r5).ins(r6)};
LSet map = LSet.empty().insAll(mapArray);

String[] colorsArray = {"Red","Blue", "Green"};
LSet colors = LSet.empty().insAll(colorsArray);

LSet x = new LSet("x");
Constraint f = x.size(3);
Ris ris = new Ris(x, map, f);
ris.output();

SolverClass solver = new SolverClass();
solver.add(regions.eq(colors));

solver.add(ris.eq(map));
solver.showStoreAll();
solver.solve();
```

4.2.2 Funzioni parziali con oggetti Ris

Un'altra caratteristica interessante è la capacità degli oggetti `Ris` di denotare *funzioni parziali*. Per avere un oggetto `Ris` che denoti una `funzione parziale`, esso dev'essere costruito con il metodo:
`public Constraint(IntLVar ct, LSet d, Constraint f, LList p)`,
dove abbiamo visto, che `p` ha la forma di una coppia ordinata di variabili

logiche intere `[x, y]`.

Esempio 7 Il seguente esempio mostra la funzione parziale denotata dall'insieme intensionale ristretto $\{x : [-10, 10] | x > -10 \bullet [x, x * x]\}$

```
IntLVar x = new IntLVar("x");
Constraint f = x.ge(-10);
LSet d = new LSet(new Interval(-10, 10).toSet());
IntLVar z = x.mul(x);
LList p = LList.empty().ins(z).ins(x);
Ris ris = new Ris(x, d, f, p);
```

Per risolvere il vincolo $[y, 25] \in \{x : [-10, 10] | x > -10 \bullet [x, x * x]\}$ basterà scrivere:

```
IntLVar y = new IntLVar("y");
LList B = LList.empty().ins(25).ins(y);
SolverClass solver = new SolverClass();
solver.add(ris.contains(B));
solver.showStoreAll();
solver.solve();
y.output();
while(solver.nextSolution()) {
  y.output();
}
```

Ed avremo come risultato:
`y = 5`
`y = -5`

Capitolo 5

RIS: aspetti implementativi

Questo capitolo descrive come è stata realizzata la classe `Ris` e come sono stati implementati i vincoli e le regole di riscrittura per i vincoli.

5.1 Campi dati Ris

La classe `Ris`, come già detto, estende la classe `LSet` e perciò eredita i suoi metodi e i suoi campi dati. I campi dati ereditati dalla classe `LSet` sono:

- `protected String name`;

- `protected boolean init`;

- `protected LCollection equ`;

- `protected Vector<Object> lista`;

- `protected LCollection resto`;

Siccome un oggetto `Ris` in `JSetL` è trattato come un unico blocco fino al momento in cui è necessario identificare almeno uno dei suoi elementi, i campi `lista` e `resto` non sono utilizzati all'interno di nessun metodo o costruttore della classe `Ris`. Essi avranno sempre il valore `null`. Si è scelto, per il momento, di non utilizzare neanche il campo `equ`, che avrà sempre valore `null`. Il campo `name` si occupa di fornire un nome esterno all'oggetto `Ris` utilizzato per l'identificazione dell'oggetto quando lo si vuole stampare su output. Il campo `init` indica se l'oggetto `Ris` è inizializzato (`true`) o meno (`false`). Il suo valore viene modificato all'interno dei costruttori, in particolare un oggetto `Ris` ha il campo `init` uguale a `true` se e solo se l'insieme che denota il suo `domain` ha anch'esso il campo `init` con il valore `true`. I campi che sono stati aggiunti alla classe `Ris` sono i seguenti:

- `protected LVar controlTerm`;

- `protected LSet domain`;

- `protected Constraint filter`;

- `protected LVar pattern`;

Il campo `controlTerm` rappresenta il *control term* dell'oggetto `Ris`, esso è di tipo `LVar` in quanto viene utilizzato come contenitore per l'oggetto che vuole essere usato come *control term*.
Ad esempio, nel costruttore `Ris(LSet set, LSet domain, Constraint f)`, il campo `controlTerm` viene inizializzato con la chiamata del costruttore `new LVar(Object o)` che imposta l'`LVar-value` dell'oggetto `controlTerm` con l'oggetto `set`. Quindi ogni volta che si ha necessità di utilizzare il campo `controlTerm` all'interno della classe, sarà utilizzato il metodo `controlTerm.value()` per estrarre l'effettivo valore del *control term* da utilizzare.
Il campo `domain`, denota il *dominio* del `Ris`, viene inizializzato assegnandogli il riferimento dell'insieme logico che si vuole utilizzare come dominio del `Ris`. Può essere un oggetto di tipo `LSet` o di tipo `Ris` (al momento, la classe Ris non supporta ancora i vincoli per insiemi intensionali che hanno per dominio un oggetto `Ris`).
Il campo `filter` è un vincolo che rappresenta il *filter* del `Ris`. Il campo `filter` coinvolge il *control Term*, perciò il filtro dovrà essere coerente con l'`LVar-value` del *control Term*.
Ad esempio, se il *control term* che si vuole usare per costruire un oggetto `Ris` è di tipo `LSet`, il vincolo `filter` non può essere un vincolo aritmetico.
Per finire, il campo `pattern` rappresenta il *pattern* del `Ris`. È anch'esso di tipo `LVar` e viene utilizzato, come per il campo `controlTerm`, come contenitore per l'oggetto che vuole essere utilizzato come *pattern*. La forma del pattern è vincolata dal tipo del *control term* che si vuole utilizzare per costruire un oggetto `Ris`.

Vediamo in dettaglio, per l'attuale versione della classe `Ris`, quali forme hanno senso per il pattern, in base al costruttore utilizzato:

- `public Ris(IntLVar ct, LSet domain, Constraint f, IntLVar p)` in questo caso il pattern può essere una qualunque *espressione logica intera* denotata da un oggetto di tipo `IntLVar`. Il pattern può essere anche semplicemente lo stesso oggetto `ct`. Ad esempio:

```
...
```

```
IntLVar pattern = x.mul(10);
Ris ris = new Ris(ct, d, f, pattern);
// {x : D | F @ __I0 = x*10 }
```

- `public Ris(IntLVar ct, LSet domain, Constraint f, LList p)`. Tale costruttore serve per creare un `Ris` della forma: $\{x : D | F \bullet (y, z)\}$. In questo caso il pattern è un oggetto di tipo `LList` la cui forma è ristretta ad essere una coppia di variabili logiche intere ordinata (y, z). Ad esempio:

```
...
IntLVar A = new IntLVar("A");
IntLVar z = x.sum(A);
LList l = LList.empty.ins(z).ins(x);
Ris ris = new Ris(x, d, f, l);
//{x : D | F @ [x, x + A]}
```

- `public Ris(LSet ct, LSet d, Constraint f)`
  `public Ris(LList ct, LSet d, Constraint f)`
  in questi due casi non è data la possibilità all'utente la possibilità di scegliere la forma del pattern, che, all'interno del costruttore, assumerà la forma del *control term* `ct`.

5.2 Metodi interni

La classe `Ris`, offre dei metodi utilizzati solo all'interno del package `JSetL`, dichiarati `protected`, che sono utili alle altri classi per poter gestire la risoluzione di vincoli.
In particolare, vedremo i metodi `F(Object d)`, `P(Obect d, Constraint p)` e `notF(Object d)` che sono metodi utili alla riscrittura dei vincoli che coinvolgono oggetti `Ris`.

5.2.1 Metodo F(Object d)

Il metodo `F(Object d)` si occupa di sostituire ogni occorrenza del campo `control Term` presente nel `Constraint filter`, con l'oggetto `d`. La necessità di avere questo metodo, nasce dalla notazione $F(d)$ vista in figura 2 del

capitolo 2. Infatti nella teoria per la riscrittura delle regole viene utilizzata questa notazione per indicare il vincolo ottenuto partendo dal vincolo F di un dato *RIS*, in cui ogni occorrenza del *control Term* è sostituita dal valore d.
La testata del metodo è la seguente:

```
protected Constraint F(Object d)
```

dove `d` è l'oggetto da sostituire al `control Term`. Restituisce un oggetto `Constraint`, che è il vincolo che si vuole ottenere.

Intuitivamente, per ottenere questo risultato il metodo `F(Object d)` funziona in questo modo:
inizialmente, scompone il `Constraint filter`, che è una congiunzione di vincoli atomici, nei suoi vincoli atomici. Per ogni vincolo atomico controlla se vi è un'occorrenza della variabile `control Term`, in questo caso la variabile sarà sostituita dall'oggetto `d`. Nel caso in cui, invece non vi è alcun occorrenza del `controlTerm`, il vincolo atomico resterà immutato.
Un caso particolare gestito dal metodo `F(Object d)` è quello in cui è presente, all'interno del campo `filter` un'*espressione logica intera*, che possieda *variabili logiche interne*, denotata da un oggetto `IntLVar`.

Ad esempio:
Sia F il vincolo $x + (y - 1) > 0$ che è il filtro per un oggetto `Ris` R, e sia x il suo *control Term*.
In `JSetL`, il vincolo F è creato dall'invocazione del metodo:

```
x.sum(y.sub(1)).gt(0).
```

Questo metodo, restituisce il vincolo `Constraint` così formato:

$$X2 > 0 \wedge X2 = X1 + x \wedge X1 = y - 1$$

dove $X2$ e $X1$ sono due variabili logiche intere *interne*. Ciò è dovuto al fatto che, come visto nel capitolo 3, la valutazione di un'*espressione logica intera*, in `JSetL`, ha una forma particolare.
□

Siccome, `F(Object d)` è un metodo che sarà utilizzato più volte all'interno dello stesso *risolutore di vincoli*, avremo che la semplice sostituzione del *control term* con un oggetto d in presenza di *espressioni logiche intere*, non avrà il risultato desiderato.

Esempio
Se chiamiamo il metodo `F(3)` e poi `F(4)` all'interno dello stesso *risolutore di vincoli* avremo un *constraint store* così formato:

$$[X2 > 0 \wedge X2 = X1 + 3 \wedge X1 = y - 1 \wedge X2 > 0 \wedge X2 = X1 + 4 \wedge X1 = y - 1]$$

(in realtà, in `JSetL`, le variabili logiche interne hanno tutte, come nome esterno di default, la stringa "_?". Sono stati utilizzati i nomi $X2, X1$ per rendere più comprensibile l'esempio).
Ma questo non è quello che vogliamo. Infatti tale congiunzione di vincoli non è equivalente all'espressione:

$$3 + (y - 1) > 0 \wedge 4 + (y - 1) > 0$$

che è ciò che ci aspettiamo dall'invocazione di $F(3)$ e $F(4)$).
Un *constraint store* equivalente all'espressione riportata di sopra, sarà di questo tipo:

$$[X2 > 0 \wedge X2 = X1 + 3 \wedge X1 = y - 1 \wedge X4 > 0 \wedge X4 = X3 + 4 \wedge X3 = y - 1]$$

□

Quindi, per ottenere un risultato coerente alla notazione $F(d)$, il metodo `F(Object d)` si occuperà anche di creare nuove variabili logiche interne nel caso in cui ci siano *espressioni logiche intere* all'interno del vincolo `filter`.
Per risolvere questo problema sono stati aggiunti due campi dati nella classe `IntLVar`:

- `public String nameInternal`, inizializzato con la stringa "__I";
- `public static int indexInternal`, inizializzato con il valore 0;

Questi campi sono usati all'interno della classe `IntLVar` ogni qualvolta che la valutazione di un'espressione logica intera genera nuove variabili logiche interne, impostando il nome di default di tali variabili con la stringa ("__I_n"), dove "__I" è il valore della stringa `nameInternal`, mentre n è il valore dell'intero `indexInternal`, che verrà incrementato ogni volta.
In questo modo `F(Object d)`, è in grado di identificare all'interno del vincolo tutte le occorrenze di variabili logiche interne (ossia quelle il cui nome inizi per "__I") e, di conseguenza, sostituirle con delle nuove. È da notare che, nel caso in cui `d` è un oggetto il cui tipo non è consistente con il vincolo `filter` sarà lanciata l'eccezione `NotDefConstraint`.

Esempio

```
package JSetL;
...
IntLVar x = new IntLVar("x");
IntLVar y = new IntLVar("y");
Constraint f =
 x.ge(0).and(x.le(10)).or(x.ge(20)).and(x.le(30))
      .or(x.mul(y).mod(6).eq(0));
// x >= 0 AND x<=10 OR x >=20 AND x<=30 OR (x * y) mod 6 = 0;
LSet d = new LSet(new Interval(-10, 10).toSet());
Ris ris = new Ris(x, d, f);
ris.F(7);
//7 >= 0 AND 7 <= 10 OR 7 >= 20 AND 7 <= 30 OR __I2 = 0 AND __I2
    = __I3 mod 6 AND __I3 = 7 * _y AND 6 neq 0
```

Notiamo che il filtro dell'oggetto `Ris` è il vincolo:

$$x \geq 0 \wedge x \leq 10 \vee x \geq 20 \wedge x \leq 30 \vee (x * y)\ mod\ 6 = 0$$

la notazione $F(7)$ (fig.2 cap. 2) denota il vincolo:

$$7 \geq 0 \wedge 7 \leq 10 \vee 7 \geq 20 \wedge 7 \leq 30 \vee (7 * y)\ mod\ 6 = 0$$

che è equivalente al `Constraint` restituito dall'istruzione `ris.F(7)`:

$$7 \geq 0 \wedge 7 \leq 10 \vee 7 \geq 20 \wedge 7 \leq 30 \vee \text{I2} = 0 \wedge \text{I2} = \text{I3 mod } 6 \wedge \text{I3} = 7 * \text{y} \wedge 6 \text{ neq } 0$$

5.2.2 Metodo notF(Object d)

Il metodo `notF(Object d)` restituisce il vincolo ottenuto dalla negazione del vincolo `Constraint filter` dove ogni occorrenza dell'oggetto `controlTerm` è sostituita dall'oggetto `d`.
Si occupa quindi di restituire il vincolo `filter` negato come nella notazione $\neg F(d)$, visto in figura 2 del capitolo 2.
La testata del metodo è la seguente:

```
protected Constraint notF(Object d)
```

dove, d è il parametro che sostituirà il `controlTerm` all'interno del `filter`, e il tipo di ritorno è `Constraint`.
Intuitivamente il metodo funziona in questo modo:
scompone il vincolo `filter` in vincoli atomici e per ognuno di essi effettua un controllo sul campo `cons` che indica il tipo di vincolo applicato. In particolare se `cons` ha il valore:

- `Environment.eqCode` che rappresenta l'operatore $=$, esso viene sostituito dal valore `Environment.neqCode` che rappresenta l'operatore $\neq$ (e viceversa);
- `Environment.leCode` che rappresenta l'operatore $\leq$, esso viene sostituito dal valore `Environment.gtCode` che rappresenta l'operatore $>$ (e viceversa);
- `Environment.ltCode` che rappresenta l'operatore $<$, esso viene sostituito dal valore `Environment.geCode` che rappresenta l'operatore $\geq$ (e viceversa);
- `Environment.inCode` che rappresenta l'operatore $\in$, esso viene sostituito dal valore `Environment.ninCode` che rappresenta l'operatore $\notin$ (e viceversa);
- `Environment.union` che rappresenta l'operatore $\cup$, esso viene sostituito dal valore `Environment.nunion` che rappresenta l'operatore $\neg\cup$ (e viceversa);
- `Environment.size` che rappresenta il vincolo size applicato ad un insieme ossia: `S.size(N)`, riscrive l'intero vincolo atomico in questo modo: `S.size(Z) AND Z.neq(N)`;

Si nota che per il momento, sono implementate le negazioni per un numero limitato di vincoli ma è facile estendere il metodo per negare altri vincoli.

Inoltre, ogni vincolo atomico, una volta negato, verrà legato con quello successivo con il vincolo $\vee$ in modo da negare anche la congiunzione per i vincoli (e viceversa). Una volta che il vincolo è stato riscritto in forma negata, viene sostituito il `controlTerm` con l'oggetto `d` come visto nella procedura `F(Object d)`.
Anche qui, in caso di presenza di *espressioni logiche intere*, è necessario un accorgimento.
Ad esempio, per come è stato descritto finora `notF(Object d)`, l'applicazione del metodo `notF(5)` quando il filtro è $x + 2 > 0$ darà come risultato il `Constraint`:

$$\texttt{5 + 2 neq \_\_I0 OR \_\_I0} \leq \texttt{0}$$

che non è il vincolo che vogliamo.
Il risultato che vogliamo è il vincolo $5+2 \leq 0$ che è equivalente al `Constraint`:

$$\texttt{5 + 2 = \_\_I0 AND \_\_I0} \leq \texttt{0}$$

□

Per ottenere ciò, è stato aggiunto il campo dati `private int iden` assieme ai suoi metodi getter e setter nella classe `AConstraint` (la classe che gestisce i vincoli atomici).
Questo campo viene modificato solamente dalla classe `IntLVar` attraverso il metodo setter `public void setIden(int x)`.
Di fatto, quando viene generata un'espressione logica intera, i vincoli atomici

$$X_1 = e_1, X_2 = e_2, ..., X_n = e_n$$

generati con essa avranno il campo dati `iden` inizializzato col valore -1. A questo punto, `notF(Object d)` effettua un ulteriore controllo sui vincoli atomici sul campo `iden`, e nel caso in cui tale campo dati abbia valore -1, il vincolo atomico resterà immutato. Inoltre, se il vincolo è in congiunzione con altri vincoli l'operatore $\wedge$ resterà anch'esso immutato (non più sostituito con l'operatore $\vee$).

Esempio

```
package JSetL;
...
IntLVar x = new IntLVar("x");
IntLVar y = new IntLVar("y");
Constraint f =x.ge(0).and(x.le(30)).or(x.mul(y).mod(6).eq(0));
// x >= 0 AND x<=30 OR (x * y) mod 6 = 0;
LSet d = new LSet(new Interval(-10, 10).toSet());
Ris ris = new Ris(x, d, f);
ris.notF(2);
//2 < 0 OR 2 > 30 AND __I4 neq 0 AND __I4 = __I5 mod 6 AND __I5 =
    2 * _y AND 6 neq 0
```

Notiamo che il filtro dell'oggetto `Ris` è il vincolo:

$$x \geq 0 \wedge x \leq 30 \vee (x * y)\ mod\ 6 = 0$$

la notazione $\neg F(2)$ (fig.2 cap. 2) denota il vincolo:

$$2 < 0 \vee 2 > 30 \wedge (2 * y)\ mod\ 6 \neq 0$$

che è equivalente al `Constraint` restituito dall'istruzione `ris.notF(2)`:

$$2 < 0 \vee 2 > 30 \wedge I4 \neq 0 \wedge I4 = I5\ mod\ 6 \wedge I5 = 2 * y \wedge 6 \neq 0$$

5.2.3 Metodo P(Object p, Constraint c)

Il metodo `P(Object p, Constraint c)` restituisce la valutazione dell'espressione denotata dal `pattern` di un oggetto `Ris`, sostituendo ogni occorrenza della variabile `controlTerm` con l'oggetto `p`.
La necessità di avere questo metodo, nasce dalla notazione $P(d)$ vista in figura 2 del capitolo 2.
La testata del metodo è la seguente:

```
protected Object P(Object d, Constraint c)
```

Il vincolo c è un parametro che funge come valore di ritorno nel caso in cui il `pattern` sia un'espressione logica intera che ha variabili logiche non inizializzate. Negli altri casi il valore di c sarà `null`.

Ad esempio, sia `y` un `IntLVar` costruita dall'espressione `x.mul(2)` e sia `x` il `controlTerm` di un oggetto `Ris R` . Inoltre, sia `y` l'espressione che denota il pattern dell'oggetto `R`. La sequenza d'istruzioni:

```
...
Constraint c = new Constraint();
IntLVar z = new IntLVar("z");
Object o = R.P(z, c);
```

porterà ad avere che:

- `o` è l'oggetto risultante la valutazione `z.sum(10)`, ossia : `I1 = unknown Domain: [-1073741823..1073741823] Constraint: __I1 = _z * 2`
- `c` sarà il `constraint` associato alla valutazione dell'espressione logica, cioè `_z*2`;

5.3 Implementazione vincoli in JSetL

Abbiamo visto che il metodo `solve()` della classe `SolverClass` implementa l'algoritmo di risoluzione dei vincoli. Il metodo `solve()` per la riscrittura dei vincoli si appoggia ad altre classi contenute nel package `JSetL`. Queste classi possiedono i metodi che si occupano effettivamente della gestione delle regole di riscrittura di vincoli. La classe `RwRulesEq` si occupa della riscrittura dei vincoli di uguaglianza e disuguaglianza delle variabili e collezioni logiche, mentre la classe `RwRulesSet` si occupa delle regole per i vincoli di appartenenza e non appartenenza ad insiemi. Nel seguito è descritto il modo in cui

sono state implementate le regole di riscrittura per i vincoli che coinvolgono oggetti `Ris`.

5.3.1 Uguaglianza

Le regole di riscrittura del vincolo di uguaglianza tra due oggetti `Ris` o un oggetto `Ris` e un oggetto `LSet` sono implementate nel metodo `eqRisLSet` della classe `RwRulesEq`. La testata del metodo `eqRisLSet` è la seguente:

```
protected void eqRisLSet(Ris arg1, LSet arg2, AConstraint s)
```

dove `arg1` e `arg2` sono rispettivamente gli oggetti `Ris` e `LSet` coinvolti nel vincolo di uguaglianza, `s` invece è l'oggetto che rappresenta il vincolo atomico di uguaglianza.
Vediamo ora l'implementazione delle regole di riscrittura in tale metodo.
La prima parte del metodo effettua un'ottimizzazione per il vincolo di uguaglianza:

```
...
LSet domain = arg1.getDomain();
if(domain.isInit() && domain.isGround() && arg2.isInit() && !(arg2
    instanceof Ris )&& arg2.isGround() && !arg2.isEmpty()) {
  if(arg1.equals(arg2)) {
    s.solved = true;
    return;
  }
  Solver.fail(s);
  return;
}
```

Infatti, in questa parte di codice di `eqRisLSet` vediamo che, se il dominio dell'oggetto `Ris` e l'oggetto `LSet` contengono solo variabili inizializzate, è invocato il metodo `equals(Object o)` sull'oggetto `Ris` ottimizzando la risoluzione del vincolo di uguaglianza. Nel caso in cui tale metodo restituisca `true` allora anche il vincolo sarà soddisfatto.
L'istruzione `s.solved = true` serve ad indicare che il vincolo atomico denotato dall'oggetto `s` di tipo `AConstraint` è risolto. In caso contrario, quando il metodo `equals` restituisce `false`, verrà eseguita l'istruzione `Solver.fail(s)`, che lancerà l'eccezione `Fail`, indicando che tale vincolo è insoddisfacibile.

Di seguito, è implementato il caso in cui sia il dominio del `Ris` che l'oggetto `LSet` sono entrambi l'insieme vuoto e allora il vincolo risulterà `true`.

$$\{x : \emptyset | F \bullet P\} = \emptyset \longrightarrow true$$

```
...
else if(arg1.isInit() && domain.isEmpty() && arg2.isInit() &&
    arg2.isEmpty()) {
  s.solved = true;
  return;
}
```

Nella seguente porzione di codice, invece, è implementata la regola di riscrittura:

$$\{x : \{d \sqcup D\} | F \bullet P\} = \emptyset \longrightarrow \neg F(d) \wedge \{x : D | F \bullet P\} = \emptyset$$

```
  ...
  else if(arg2.isInit() && arg2.isEmpty() && arg1.isInit()) {
  Ris ris = new Ris(arg1.getControlTerm(), domain.removeOne(),
      arg1.getFilter(), arg1.getPattern());
  s.arg1 = ris;
  Solver.add(arg1.notF(domain.getOne()));
  Solver.storeInvariato = false;
  return;
}
```

L'istruzione `s.arg1 = ris`, modifica il vincolo atomico `s`, già presente nel `constraint solver`, cambiando il valore del primo argomento con l'oggetto `ris`, che denota il *RIS*: $\{x : D | F \bullet P\}$.
Il metodo `Solver.add(arg1.notF(domain.getOne()))`, invece, aggiunge al `constraint store` del *risolutore di vincoli* `Solver` il vincolo generato dal metodo `notF(d)`.

Qui vediamo altre due ottimizzazioni:

```
...
else if(arg1.isInit() && (arg2.isInit() && !arg2.isEmpty() ||
    !arg2.isInit())) {
  //case : {{} | F @ P} = B ^ B is initialized ----> FALSE
  if(domain.isEmpty() && arg2.isInit()) {
    if(!arg2.isEmpty()) {
```

```
      Solver.fail(s);
      return;
    }
  }
  //case : {{} | F @ P} = B ^ B isn't initialized ----> B := {}
  else if(domain.isEmpty()) {
    arg2.setInit(true);
    arg2.setEqu(LSet.empty());
    s.solved = true;
    Solver.storeInvariato = false;
    return;
  }
```

La prima gestisce il caso in cui il dominio del `Ris` è l'insieme vuoto, mentre l'oggetto `LSet` è un insieme inizializzato non vuoto. In questo caso il vincolo verrà riscritto in `false`.
La seconda è il caso in cui il dominio del `Ris` è l'insieme vuoto, mentre l'oggetto `LSet` è un insieme non inzializzato, allora tale oggetto sarà inizializzato come l'insieme vuoto, e il vincolo sarà di conseguenza soddisfatto.

Di seguito è mostrata la regola di riscrittura:

$$\{x : \{d \sqcup D\} | F \bullet P\} = B \longrightarrow$$
$$F(d) \wedge \{P(d) \sqcup \{x : D | F \bullet P\} = B \vee$$
$$\neg F(d) \wedge \{x : D | F \bullet P\} = B$$

```
...
switch(s.caseControl) {
  //----> F(d) ^ {P(d) u {D| F @ P}} = B
  case 0:
  Solver.addChoicePoint(s);
  Solver.add(arg1.F(domain.getOne()));
  LSet ris = new Ris(arg1.getControlTerm(), domain.removeOne(),
      arg1.getFilter(), arg1.getPattern());

  if(domain.removeOne().isGround() &&
      !domain.removeOne().isEmpty()) {
    ris = ((Ris) ris).espandi();
  }

  Constraint rc = new Constraint();
```

```
    Object r = arg1.P(domain.getOne(), rc);
    if(!rc.isEmpty()) Solver.add(rc);

    LSet ins = LSet.empty();
    if(domain.removeOne().isEmpty()) {
      ins = LSet.empty().ins(r);
      s.arg1 = ins;
      return;
    }
    else {
      ins = new LSet(LSet.empty().ins(r).toVector(), ris);
    }
    s.arg1 = ins;
    Solver.storeInvariato = false;
    return;
    //----> not(F(d)) ^ {D | F @ P } = B
    case 1:
    Solver.add(arg1.notF(domain.getOne()));
    Ris ris1 = new Ris(arg1.getControlTerm(), domain.removeOne(),
        arg1.getFilter(), arg1.getPattern());
    s.arg1 = ris1;
    s.caseControl = 0;
    Solver.storeInvariato = false;
    return;
  }
}
```

L'istruzione `Solver.addChoicePoint(s)` aggiunge un *punto di scelta* per trattare il non determinismo e il backtracking di questa regola.
È da notare la parte di codice:

```
if(domain.removeOne().isGround() && !domain.removeOne().isEmpty())
{
  ris = ((Ris) ris).espandi();
}
```

Questa è un'ulteriore ottimizzazione che avviene nel caso in cui la parte restante del `domain` del `Ris` contenga solo variabili logiche inizializzate. In questo caso, il `Ris` passa alla sua forma estensionale attraverso il metodo `espandi` restituendo un oggetto `LSet`, e perciò la restante risoluzione del vincolo sarà affidata alle metodo `eqSet(LSet s1, LSet s2, AConstraint`

`s`) che si occupa delle regole di riscrittura per il vincolo di uguaglianza tra insiemi estensionali. Per finire, questa parte di codice implementa la regola:

$$\bar{D} = \{z_{\sqcup} E\} \wedge F(z) \wedge y = P(z) \wedge \{x : E | F \bullet P\} = A$$

```
else if(arg2.isInit() && !arg2.isEmpty() && !arg1.isInit()) {
  LSet C = new LSet("C");
  Object d = new Object();
  LSet G = new LSet("G");
  if(arg1.getControlTerm().value() instanceof IntLVar) {
    d = new IntLVar();
  }
  else if(arg1.getControlTerm().value() instanceof LSet) {
    d = new LSet();
  }
  Solver.add((C.ins(d)).eq(domain));
  Solver.add(domain.eq(G));
  Solver.add(arg1.F(d));
  Constraint rc = new Constraint();
  Object r = arg1.P(d, rc);
  Solver.add(new Constraint(arg2.getOne(), Environment.eqCode,
     r));
  if(!rc.isEmpty()) Solver.add(rc);
  s.arg1 =new Ris(arg1.getControlTerm(), C, arg1.getFilter(),
     arg1.getPattern());
  s.arg2 = arg2.removeOne();
  return;
}
```

5.3.2 Disuguaglianza

Le regole di riscrittura del vincolo di disuguaglianza tra due oggetti `Ris` o un oggetto `Ris` e un oggetto `LSet` sono implementate nel metodo `neqRis` della classe `RwRulesEq`. La testata del metodo `neqRis` è la seguente:

```
protected void neqRis(Ris ris, LSet set, AConstraint s)
```

dove `arg1` e `arg2` sono risperttivamente gli oggetti `Ris` e `LSet` coinvolti nel vincolo di disuguaglianza, `s` è, invece, l'oggetto che rappresenta il vincolo atomico di disuguaglianza. E questa è l'implementazione:

```
    LSet domain = ris.getDomain();

  //{{} | F(x) @ P(x)} != {} ----> FALSE
  if(domain.isInit() && set.isInit() && domain.isEmpty() &&
      set.isEmpty()) {
    Solver.fail(s);
    return;
  }
  //{{} | F(x) @ P(x)} != A ^ A isn't Empty ----> TRUE
  else if(domain.isInit() && set.isInit() && domain.isEmpty() &&
      !set.isEmpty()) {
    s.solved = true;
    return;
  }
  //{x in D | F(x) @ P(x)} != A
  else {
    switch(s.caseControl) {
      case 0:
      // ----> y in {D | F @ P} ^ y nin A
      Solver.addChoicePoint(s);
      LVar y = new LVar("y");
      s.arg2 = ris;
      s.arg1 = y;
      s.cons = Environment.inCode;

      Solver.add(y.nin(set));
      Solver.showStoreAll();
      Solver.storeInvariato = false;
      return;

      case 1:
      // ----> y nin {D | F @ P} ^ y in A
      s.caseControl = 0;
      LVar y1 = new LVar();
      s.arg2 = ris;
      s.arg1 = y1;
      s.cons = Environment.ninCode;

      Solver.add(y1.in(set));

      Solver.storeInvariato = false;
      return;
```

```
    }
  }
}
```

5.3.3 Appartenenza

Le regole di riscrittura del vincolo di appartenenza di una variabile logica in un `Ris` sono implementate nel metodo `inLVarRis` della classe `RwRulesSet`. La testata del metodo è la seguente:

```
protected void inLVarRis(LVar var, Ris ris, AConstraint s)
```

dove `var` e `ris` sono rispettivamente gli oggetti `LVar` e `Ris` coinvolti nel vincolo di appartenenza, `s` è, invece, l'oggetto che rappresenta il vincolo atomico di appartenenza.

E questa è l'implementazione:

```
LSet domain = ris.getDomain();
//y in {x in {} | F(x) @ P(x)} --> FALSE
if(domain.isInit() && domain.isEmpty()) {
  Solver.fail(s);
  return;
}
else if(!domain.isInit()) {
  IntLVar d = new IntLVar();
  s.arg1 = d;
  s.arg2 = domain;
  Solver.add(ris.setFilterControlTerm(d));
  Constraint rp = new Constraint();
  Object p = ris.P(d, rp);
  if(!rp.isEmpty()) Solver.add(rp);
  Solver.add(lvar.eq(p));
  Solver.storeInvariato = false;
  return;
}
else if(domain.isInit()) {
  switch(s.caseControl) {
    case 0:
    Solver.addChoicePoint(s);
    IntLVar d = new IntLVar();
```

```
      if(domain.getOne() instanceof Integer) {
        d = new IntLVar((Integer) domain.getOne());
      }
      else if(domain.getOne() instanceof IntLVar) {
        d = new IntLVar((IntLVar) domain.getOne());
      }
      Ris D = new Ris(ris.getControlTerm(), domain.removeOne(),
          ris.getFilter(), ris.getPattern());
      Solver.add(ris.setFilterControlTerm(d));
      Constraint rp = new Constraint();
      Object p = ris.P(d, rp);
      if(!rp.isEmpty()) Solver.add(rp);
      LSet ins = new LSet(LSet.empty().ins(p).toVector(), D);
      s.arg2 = ins;
      Solver.storeInvariato = false;
      return;
      case 1:
      s.caseControl = 0;
      IntLVar d1 = new IntLVar();
      if(domain.getOne() instanceof Integer) {
        d1 = new IntLVar((Integer) domain.getOne());
      }
      else if(domain.getOne() instanceof IntLVar) {
        d1 = new IntLVar((IntLVar) domain.getOne());
      }
      Ris D1 = new Ris(ris.getControlTerm(), domain.removeOne(),
          ris.getFilter(), ris.getPattern());
      Solver.add(ris.notSetFilterControlTerm(d1));
      s.arg2 = D1;
      Solver.storeInvariato = false;
      return;
    }
  }
}
```

Notiamo che anche in questo caso, per implementare il backtracking e il non determinismo, viene aggiunto un punto di scelta con l'istruzione:
`Solver.addChoicePoint(s);`

5.3.4 Non appartenenza

Le regole di riscrittura del vincolo di non appartenenza di una variabile logica in un `Ris` sono implementate nel metodo `ninLVarRis` della classe `RwRulesSet`. La testata del metodo è la seguente:

```
protected void ninLVarRis(LVar lvar, Ris ris, AConstraint s)
```

dove `lvar` e `arg2` sono rispettivamente gli oggetti `LVar` e `Ris` coinvolti nel vincolo di non appartenenza, `s` è, invece, l'oggetto che rappresenta il vincolo atomico di non appartenenza. E l'implementazione è fatto in questo modo:

```
LSet domain = ris.getDomain();
//y nin {} ----> TRUE
if(domain.isInit() && domain.isEmpty()) {
  s.solved = true;
  return;
}
else if(domain.isInit()) {
  switch(s.caseControl) {
    case 0:
    Solver.addChoicePoint(s);
    Ris D = new Ris(ris.getControlTerm(), domain.removeOne(),
       ris.getFilter(), ris.getPattern());
    Object d = domain.getOne();
    Constraint c = new Constraint();
    Object p = ris.P(d, c);
    if(!c.isEmpty()) Solver.add(c);
    Solver.add(ris.F(d));
    Solver.add(lvar.neq(p));
    s.arg2 = D;
    Solver.storeInvariato = false;
    return;
    case 1:
    s.caseControl = 0;
    Object d1 = domain.getOne();
    Ris D1 = new Ris(ris.getControlTerm(), domain.removeOne(),
       ris.getFilter(), ris.getPattern());
    Solver.add(ris.notF(d1));
    s.arg2 = D1;
    return;
  }
}
```

Capitolo 6

Conclusioni e sviluppi futuri

In questo lavoro di tesi è stata implementata un'estensione della libreria Java `JSetL`, che consenta di gestire gli *insiemi intensionali ristretti* e i relativi vincoli associati. Tale estensione è stata effettuata seguendo la descrizione del linguaggio $\mathcal{L}_{\mathcal{RIS}}$ in [] e in particolare seguendo le regole di riscrittura dei vincoli associati. Al momento `JSetL` possiede la nuova classe `Ris` che consente di trattare oggetti che rappresentano gli *insiemi intensionali ristretti*. Oltre alla classe `Ris`, sono state implementate le regole di riscrittura dei vincoli di:

- uguaglianza $=$;
- disuguaglianza $\neq$;
- appartenenza $\in$;
- non appartenenza $\notin$;

Per effettuare ciò, sono state fatte delle modifiche alle classi `RwRulesEq` e `RwRulesSet` che si occupano di definire le regole di riscrittura per i vincoli di uguaglianza e di appartenenza e le loro negazioni.
Grazie alla risoluzione del vincolo di uguaglianza è stato possibile implementare l'unificazione anche per i *RIS*.
Non si è tenuto conto dell'efficienza del processo di risoluzione di vincoli, in quanto lo scopo principale della libreria `JSetL` è quello di supportare la programmazione *dichiaritiva* all'interno del linguaggio Java, attraverso l'uso di strutture dati come variabili logiche, liste e insiemi, ma anche attraverso l'unificazione e la risoluzione di vincoli basata sul non determinismo e sul backtracking.
Abbiamo visto anche la capacità espressiva dei *RIS* ed alcune loro applicazioni d'uso come la rappresentazione di *quantificatori universali ristretti* e *funzioni parziali*.

Per quanto riguarda gli sviluppi futuri sarebbe interessante aggiungere alla classe `Ris` la possibilità di gestire ulteriori vincoli e di conseguenza implementarne le regole di riscrittura. In particolare, siano `s1`, `s2` ed `s3` tre insiemi logici. Sarebbe interessante estendere la classe `Ris` con i vincoli di:

- unione `s1.union(s2, s3)`, dove almeno uno dei tre operandi sia un oggetto di tipo `Ris`;
- intersezione `s1.inters(s2, s3)` dove almeno uno dei tre operandi sia un oggetto di tipo `Ris`;
- complementazione di insiemi intensionali;
- disgiunzione `s1.disj(s) dove almeno un operando sia un oggetto di tipo Ris`;
- differenza `s1.diff(s2, s3)`; dove almeno uno dei tre operandi sia un oggetto di tipo `Ris`;
- inclusione `s1.subset(s2)`; dove almeno uno dei due operandi sia un oggetto di tipo `Ris`.

Bibliografia

[1] 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

[2] Maximiliano Cristiá and Gianfranco Rossi.
A Decision Procedure for Restricted Intensional Sets.
In Automated Deduction – 26th International Conference on Automated Deduction (CADE 26), de Moura, Leonardo (Ed.), Lecture Notes in Artificial Intelligence, Vol. 10395, Springer International Publishing, pages: 185-201, 2017

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

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

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