Indice

Capitolo 1

Introduzione

Recentemente si è cercato di far coesistere la progammazione dichiarativa, in particolare quella basata su vincoli, con i tradizionali linguaggi di programmazione OO (Object-Oriented), per dotare questi linguaggi di alto livello, come C++ e Java, dei principali vantaggi della DP (cioè Declarative Programming), in cui si pone l'enfasi sul *"che cosa"* si deve risolvere piuttosto che sul *"come"* risolvere un problema. In questo modo vengono definiti nuovi linguaggi oppure alcune estensioni dei linguaggi esistenti.

Invece di estendere un linguaggio convenzionale per supportare la programmazione con vincoli, definendo una nuova libreria abbiamo come esempi le librerie JSolver, Choco, Koalog, JACK e JCL (Java Constraint Library) in Java e ILOG SOLVER in C++, un sistema in cui le variabili logiche e i vincoli (constraint) si comportano come oggetti e sono definiti in una classe di libreria in linguaggio C++. Tutte queste proposte hanno concentrato la loro attenzione sull'espressività dei vincoli e sull'efficienza del processo di risoluzione. Questo approccio basato su libreria evidenzia come sfruttare gli importanti vantaggi offerti dai linguaggi di programmazione orientata agli

oggetti, ovvero definire nuovi tipi e nuove operazioni, mantenendo una buona separazione tra interfaccia ed implementazione.

In questo contesto si inserisce JSetL, una libreria implementata in Java [1]. Essa estende il paradigma della programmazione orientata agli oggetti con alcune caratteristiche fondamentali dei linguaggi CLP (Constraint Logic Programming): l'uso di variabili logiche, l'unificazione, il non-determinismo e la risoluzione di vincoli. Questa libreria è basata sul linguaggio CLP($\mathcal{SET}$) [6], in cui gli insiemi sono la struttura privilegiata e gli algoritmi di risoluzione dei vincoli sono quelli di $SAT_{\mathcal{SET}}$. I vincoli fino ad ora introdotti in JSetL sono principalmente di tipo insiemistico: uguaglianza, disuguaglianza, appartenenza, unione, disgiunzione ed altre operazioni quali intersezione, inclusione...

Oltre agli insiemi, in informatica, i multi-insiemi rivestono un ruolo importante. I multi-insiemi sono una delle strutture dati che più di frequente si rivelano particolarmente utili nell'ambito di costruzione di programmi. Essi vengono chiamati anche *bag* e sono caratterizzati dalla presenza della proprietà permutativa

$$f(x, f(y, z)) = f(y, f(x, z))$$

e dall'assenza di quella di assorbimento

$$f(x, f(x, y)) = f(x, y)$$

che invece è presente per gli insiemi. La differenza fra insiemi e multi-insiemi consiste quindi nella proprietà di assorbimento; perciò in un insieme non è rilevante il numero di occorrenze di un elemento, mentre lo è per un multi-

insieme.

Nonostante l'utilità dei multi-insiemi in ambiente informatico, pochissimi linguaggi prevedono la gestione dei multi-insiemi. Fra questi si ricorda il C++, che li fornisce tramite la libreria standard, e il SICSTUS PROLOG (tramite la procedura `bagof`) [11]. Di recente è stato sviluppato SETA, un linguaggio che combina programmazione funzionale e programmazione logica, in grado di gestire vincoli su multi-insiemi [8]. Si nota però l'assenza di un linguaggio che fornisca insiemi e multi-insiemi come strutture privilegiate inserendole in una trattazione uniforme, di cui abbiamo una visione assiomatica nell'articolo di A. Dovier, A. Policriti e G. Rossi [4]. L'importanza dei multi-insiemi è stata sottolineata da diversi autori, tra cui J.P. Banâtre e D. Le Métayer che svilupparono GAMMA, cioè **G***eneral* **A***bstract* **M***odel for* **M***ultiset* **M***anipulation*, ovvero un modello di programmazione parallela basato sulla trasformazione di multi-insiemi [9]. I multi-insiemi sono oggetto di importanti applicazioni come ad esempio Chemical Abstract Machine [12]. Altre applicazioni interessanti che sfruttano la riscrittura dei multi-insiemi si trovano nei P-systems [13] e in sistemi per la gestione dei database.

Data quindi l'importanza dei multi-insiemi in ambito informatico e non solo, si è voluto estendere la libreria Java JSetL con l'introduzione della nuova struttura dati multi-insieme. L'estensione è avvenuta sulla base del linguaggio CLP($\mathcal{SET}$+$\mathcal{BAG}$), un'istanza dello schema CLP che incorpora sia insiemi che multi-insiemi. I vincoli introdotti per i multi-insiemi sono uguaglianza, disuguaglianza, appartenenza semplice e non appartenenza.

L'estensione che si vuole ottenere dovrà consentire all'utente di manipolare oggetti di tipo multi-insieme, nel rispetto delle proprietà che questa

nuova struttura dati possiede. I multi-insiemi potranno essere annidati l'uno dentro l'altro e parzialmente specificati, potranno contenere variabili logiche o altri termini (a loro volta contenenti variabili). L'introduzione dei multi-insiemi non dovrà influenzare in alcun modo la gestione degli aggregati già presenti in JSetL, ovvero liste e insiemi; anzi, i multi-insiemi devono poter coesistere con le strutture dati esistenti. In questo modo i multi-insiemi nascono in JSetL come una struttura dati indipendente, ma allo stesso tempo in grado di annidarsi a strutture di altra natura.

Come accade per tutti gli aggregati di JSetL, anche per i multi-insiemi vengono fissate alcune operazioni fondamentali, precisamente $\in$, $\notin$, $\neq$ e $=$, che, come al solito sono trattati come vincoli. Il meccanismo di risoluzione e riscrittura dei vincoli non verrà cambiato in seguito all'introduzione di questa nuova struttura dati, anzi, le nuove regole introdotte permetteranno di riscrivere e semplificare i vincoli sempre più fino ad arrivare alla consueta forma risolta, oppure ad un fallimento, anche in presenza di multi-insiemi.

La dissertazione è organizzata come segue:

Capitolo 2 Una visione assiomatica dei multi-insiemi e le regole di riscrittura di vincoli su multi-insiemi sono l'oggetto di questo capitolo.

Capitolo 3 In questo capitolo viene presentata la libreria JSetL: le caratteristiche principali delle sue strutture, il meccanismo di risoluzione dei vincoli e il non-determinismo.

Capitolo 4 Questo capitolo illustra i multi-insiemi e i vincoli che possono coinvolgere questa nuova struttura da un punto di vista dell'utente che utilizza la libreria JSetL.

Capitolo 5 In questo capitolo viene descritta l'implementazione in JSetL della classe MultiSet, dei vincoli su multi-insiemi e le modifiche che sono state necessarie per permettere la coesistenza in JSetL di questa nuova struttura e di variabili logiche, liste ed insiemi.

Capitolo 6 Alcuni semplici esempi in cui si mostra come sfruttare i multi-insiemi e la risoluzione di vincoli attraverso un interprete GAMMA, realizzato sfruttando JSetL.

Capitolo 2

Vincoli su multi-insiemi in CLP($\mathcal{BAG}$).

La Programmazione Logica a Vincoli (in inglese *Constraint Logic Programming*, ossia CLP) nasce come unione fra risoluzione di Vincoli e Programmazione Logica, estendendole; è per questo che i linguaggi CLP risultano particolarmente flessibili ed espressivi. La progammazione con vincoli costituisce una tecnica relativamente nuova di programmazione che presenta diversi interessanti aspetti sia dal punto di vista teorico, che metodologico, che applicativo. I linguaggi logici sono perfetti per trattare con vincoli e meccanismi per la loro gestione, anche se offrono modalità di programmazione piuttosto distanti da quelle della programmazione tradizionale.

2.1 Nozioni preliminari

Prima di parlare di risoluzione di vincoli su multi-insiemi in CLP($\mathcal{BAG}$) è necessario dare alcune nozioni preliminari e alcune notazioni.

Definizione 1 (Linguaggio del primo ordine). *Un* **linguaggio del primo ordine** $\mathcal{L} = \langle \Sigma, \mathcal{V} \rangle$ *è definito dall'alfabeto* $\Sigma = \langle \mathcal{F}, \Pi \rangle$*, composto dall'insieme* $\mathcal{F}$ *di simboli di costanti e funzioni, dall'insieme* Π *di simboli di predicato, e dall'insieme non numerabile* $\mathcal{V}$ *di variabili.*

Definizione 2 (Teoria del primo ordine). *Una* **teoria del primo ordine** $\mathcal{T}$ *su un linguaggio* $\mathcal{L}$ *è un insieme di formule chiuse del primo ordine di* $\mathcal{L}$ *tali che, ogni formula chiusa di* $\mathcal{L}$ *che può essere dedotta da* $\mathcal{T}$*, è in* $\mathcal{T}$*.*

Definizione 3 (Insieme di assiomi del primo ordine). *Un* **insieme di assiomi del primo ordine** Θ *su un linguaggio* $\mathcal{L}$ *è un insieme di formule chiuse del primo ordine di* $\mathcal{L}$*. Un insieme di assiomi* Θ *si chiama anche* **assiomatizzazione** *di* $\mathcal{T}$*, se* $\mathcal{T}$ *è la più piccola teoria che contiene* Θ*, cioè tale che* $\Theta \subseteq \mathcal{T}$*.*

Definizione 4 (Assioma di equivalenza e teoria di equivalenza). *Un* **assioma di equivalenza** *è una formula del tipo* $\forall (l = r)$ *dove* l *ed* r *sono termini. Una teoria di equivalenza* E *è un'assiomatizzazione i cui assiomi sono assiomi di equivalenza. Dati due termini* l *ed* r*, possiamo scrivere* $l \approx_E r$ *, se possiamo dimostrare che* l *è equivalente ad* r *usando gli assiomi di* E*.*

2.2 Teoria e vincoli su multi-insiemi

In questo capitolo analizzeremo, in primo luogo la teoria assiomatica dei multi-insiemi, poi, nella seconda parte, ci concentreremo sulla procedura di riscrittura dei vincoli su multi-insiemi.

Il linguaggio $\mathcal{L}_{MSet}$ è definito come $\langle \Sigma_{MSet}, \mathcal{V} \rangle$, dove $\mathcal{V}$ è l'insieme delle variabili e $\Sigma_{MSet} = \langle \mathcal{F}_{MSet}, \Pi \rangle$, in cui $\mathcal{F}_{MSet}$ contiene $\{\!| \cdot | \cdot |\!\}$, costruttore della

struttura multi-insieme, e la costante `nil`, che denota invece il multi-insieme vuoto, mentre $\Pi = \{=, \in\}$.

Definizione 5 *(***Vincolo***) Un vincolo è la congiunzione di formule atomiche o la negazione di formule atomiche del linguaggio $\mathcal{L}_{MSet}$, del tipo $s\pi t$, dove $\pi \in \Pi$, e s, t sono termini di $\mathcal{F}_{MSet}$.*

Esempio 1 *I seguenti sono esempi di vincoli su multi-insiemi:*

1. $X \in \{\!\{ a, b | Y \}\!\} \wedge X \notin Y \wedge X \neq a$;

2. $\{\!\{ \{\!\{ h, h, o \}\!\}, \{\!\{ o, o \}\!\} \}\!\} = \{\!\{ W, O \}\!\} \wedge Y_1 \in W \wedge Y_2 \in W \wedge Y_1 \neq Y_2$;

3. $\{\!\{ e, e, e, e, e \}\!\} = \{\!\{ X | R \}\!\} \wedge X \notin \{\!\{ e | S \}\!\}$,

ricordando che le lettere maiuscole indicano le variabili, mentre le lettere minuscole rappresentano i simboli di costanti.

Ci occupiamo ora di illustrare la teoria del primo ordine relativa ai multi-insiemi, chiamata $\mathcal{MS}$et. Essa è regolata dai seguenti assiomi:

(K)	$\forall X, Y_1, \ldots, Y_n$ $f \in \mathcal{F}_{MSet}$ *e* f *non è* $\{\!\{ \cdot \mid \cdot \}\!\}$	$(X \notin f(Y_1, \ldots, Y_n))$
(W)	$\forall Y, V, X$	$(X \in \{\!\{ Y \mid V \}\!\} \leftrightarrow X \in V \vee X = Y)$
(F_1)	$\forall X_1, \ldots, X_n, Y_1, \ldots, Y_n$ $f \in \mathcal{F}_{MSet}$	$\left(\begin{array}{r} f(X_1, \cdots X_n) = f(Y_1, \cdots, Y_n) \\ \rightarrow X_1 = Y_1 \wedge \ X_n = Y_n \end{array} \right)$
(F_2)	$\forall X_1, \ldots, X_n, Y_1, \ldots, Y_n$ $f, g \in \mathcal{F}_{MSet}$ *e* f *non è* g	$f(X_1, \cdots \ X_n) \neq g(Y_1, \cdots, \ Y_n)$
(F_3^m)	$\forall \ X_1, \ldots, X_m,$ $Y_1, \ldots, Y_n, X$	$\left(\begin{array}{l} \{\!\{ X_1, \ldots, X_m \mid X \}\!\} = \{\!\{ Y_1, \ldots, Y_n \mid X \}\!\} \\ \rightarrow \{\!\{ X_1, \ldots, X_m \}\!\} = \{\!\{ Y_1, \ldots, Y_n \}\!\} \end{array} \right)$
(E_k^m)	$\forall Y_1, Y_2, V_1, V_2,$	$\left(\begin{array}{l} \{\!\{ Y_1 \mid V_1 \}\!\} = \{\!\{ Y_2 \mid V_2 \}\!\} \leftrightarrow \\ (Y_1 = Y_2 \wedge V_1 = V_2) \vee \\ \exists Z \ (V_1 = \{\!\{ Y_2 \mid Z \}\!\} \wedge \ (V_2 = \{\!\{ Y_1 \mid Z \}\!\})) \end{array} \right)$

La teoria $\mathcal{MS}et$ è descritta dagli assiomi precedentemente illustrati: (K), (W) e (F_1), (F_2), che rappresentano gli assiomi di uguaglianza di Clark, (F_3^m) che formalizza la proprietà di aciclicità dei multi-insiemi, ovvero non possono esistere termini che siano sottotermini di se stessi ed infine (E_k^m) che enuncia la permutatività degli elementi in un multi-insieme.

Quest'ultimo assioma è di fondamentale importanza, in quanto afferma che due multi-insiemi sono uguali se e solo se essi hanno lo stesso numero di occorrenze dello stesso elemento, indipendentemente dall'ordine in cui questi compaiono.

L'assioma (E_k^m) implica la permutatività degli elementi su cui si basa la teoria di equivalenza $E_{\mathcal{MS}et}$, costituita dalla seguente uguaglianza:

(E_p^m)	$\forall X, Y, Z$	$\{\!\{ X, Y \mid Z \}\!\} = \{\!\{ Y, X \mid Z \}\!\}$

Esiste un modello di interpretazione privilegiato $\mathcal{BAG}$, che viene utilizzato per testare la soddisfacibilità dei vincoli; questo è possibile, in quanto si può dimostrare che la teoria e il rispettivo modello sono corrispondenti sulle classi di vincoli. Definiamo $\mathcal{BAG}$ come partizionamento del'Universo di Herbrand ottenuto tramite $E_{\mathcal{MS}et}$:

Definizione 6 *Il modello $\mathcal{BAG}$ è costruito nel seguente modo:*

1. *Sia $\mathcal{T}(\mathcal{F}_{\mathcal{MS}et})$ la teoria del primo ordine associata ai multi-insiemi, costruita a partire dai simboli contenuti in $\mathcal{F}_{\mathcal{MS}et}$;*

2. *Il dominio del modello è il quoziente* $\mathcal{T}(\mathcal{F}_{\mathcal{MS}et})/\equiv_{E_{\mathcal{MS}et}}$ *dell'universo di Herbrand* $\mathcal{T}(\mathcal{F}_{\mathcal{MS}et})$ *sulla più piccola relazione di congruenza* $\equiv_{E_{\mathcal{MS}et}}$ *indotta dalla teoria di equivalenza* $E_{\mathcal{MS}et}$ *su* $\mathcal{T}(\mathcal{F}_{\mathcal{MS}et})$*;*

3. *L'interpretazione di un termine* t *è la sua classe di equivalenza* $[t]$*;*

4. *Il simbolo* $'='$ *ha il significato di funzione identità sul dominio* $\mathcal{T}(\mathcal{F}_{\mathcal{MS}et})/\equiv_{E_{\mathcal{MS}et}}$*;*

5. *L'interpretazione del simbolo di appartenenza* $'\in'$ *è la seguente:* $[t] \in [s]$ *è* **true** *se e solo se in* $[s]$ *esiste un termine della forma* $\{\!|\, t_1, \ldots, t_n, t \mid r \,|\!\}$ *per alcuni termini* $t_1, \ldots, t_n, t, r$*.*

In generale, data una teoria $\mathcal{T}$ su un linguaggio del primo ordine $\mathcal{L}$ e un modello $\mathcal{A}$, $\mathcal{T}$ e $\mathcal{A}$ *si corrispondono* su un insieme di vincoli ammissibili $\mathcal{A}$dm se, per ogni vincolo $C \in \mathcal{A}$dm, si ha $\mathcal{T} \models \overrightarrow{\exists}(C)$ se e solo se $\mathcal{A} \models \overrightarrow{\exists}(C)$.

Si dimostra che questo vale anche per la teoria $\mathcal{MS}et$ e il modello $\mathcal{BAG}$:

Teorema 1 (Corrispondenza) *Il modello* $\mathcal{BAG}$ *e la teoria Bag si corrispondono sulla classe dei vincoli ammissibili.*

Questo teorema garantisce che se un vincolo è soddisfatto (rispettivamente insoddisfatto) nel modello, allora lo è anche nella teoria del primo ordine corrispondente.

La nozione di *soddisfacibiltà* è strettamente legata alla nozione di *valutazione* e viceversa.

Definizione 7 (Valutazione). *Una* **valutazione** σ *è una funzione da un sottoinsieme di* $\mathcal{V}$ *in* A*. Ogni valutazione può essere estesa ad una funzione da* $\mathcal{T}(\mathcal{F}, \mathcal{V})$ *in* A *e ad una funzione da un insieme di formule su un linguaggio* $\mathcal{L}$ *all'insieme* {**true**,**false**}*. Una valutazione* σ *si dice* **valutazione con successo** *di* φ *se* $\sigma(\varphi) =$ **true**.

Esempio 2 *Consideriamo il vincolo:*
$a \in X \wedge B \in X \wedge X = \{\!\!\{ Y, Z \}\!\!\} \wedge B \neq a.$
La valutazione con successo, qui di seguito riportata, assicura che tale vincolo sia soddisfatto in $\mathcal{BAG}$ (e per il teorema 1 in tutti i modelli della teoria relativa ai Bag):

$B \mapsto \mathbf{nil}$ $\qquad$ $Y \mapsto a$
$X \mapsto \{\!\!\{ a, \mathbf{nil} \}\!\!\}$ $\qquad$ $Z \mapsto \mathbf{nil}$

Consideriamo ora un altro vincolo:
$a \in X \wedge B \in X \wedge X = \{\!\!\{ Y \}\!\!\} \wedge B \neq a.$
non è soddisfacibile in $\mathcal{BAG}$ (e per il teorema 1 neanche in altri modelli).

□

2.3 Risoluzione di vincoli su multi-insiemi

Per controllare la soddisfacibiltà dei vincoli sui multi-insiemi in CLP($\mathcal{BAG}$) si sfrutta il meccanismo di risoluzione dei vincoli, in cui ogni vincolo C, se risulta soddisfacibile, viene ridotto in modo non-deterministico in una forma speciale, chiamata forma risolta, altrimenti C verrà ridotto a `false`. Dato che la *forma risolta* gioca un ruolo fondamentale nel constraint solving, ne diamo una definizione formale:

Definizione 8 (Solved Form). *Un vincolo atomico C è in forma risolta se è uno dei seguenti:*

- *$X = t$, con X che non occorre nè in t, nè nel resto del vincolo;*
- *$X \neq t$, con X che non occorre in t;*

- $t \notin X$, *con* X *che non occorre in* t.

Un vincolo C *è in forma risolta se è vuoto o se tutti i suoi vincoli atomici sono contemporaneamente in forma risolta.*

Osserviamo che per i multi-insiemi vale che:

$$\mathrm{s} \in \mathrm{t} \leftrightarrow \exists N\ (\mathrm{t} = \{\!\!\{\ \mathrm{s} \mid N\ \}\!\!\}).$$

Ovvero, nei multi-insiemi tutti i vincoli di appartenenza possono venire sostituiti da equivalenti vincoli di uguaglianza.

Tutte le soluzioni di un vincolo C si possono ottenere nel seguente modo:

$SAT_{\mathcal{B}ag}(C) =$	repeat
	$C' := C$;
	$C := Unify\_bags(neq\_bag(nin\_bag(in\_bag(C))))$;
	until $C = C'$;
return C.	

Figura 2.1: Procedura $SAT_{\mathcal{B}ag}$

La procedura non-deterministica $SAT_{\mathcal{B}ag}$ verifica la soddisfacibilità di un vincolo C nella teoria $\mathcal{MS}et$ e usa in modo iterativo i vari meccanismi di riscrittura dei vincoli per ridurre il vincolo C in forma risolta. $SAT_{\mathcal{B}ag}(C)$ restituisce una collezione $C_1, \ldots, C_k$ di vincoli, ognuno dei quali è in forma risolta oppure `false`. Ognuno dei C_i, in forma risolta, rappresenta implicitamente una famiglia di soluzioni per C.

Il teorema che segue garantisce la soddisfacibilità di un vincolo in forma risolta nel modello $\mathcal{BAG}$:

Teorema 2 (Soddisfacibilità). *Sia* C *un vincolo in forma risolta nel linguaggio* $\mathcal{L}_{Bag}$*, esso è soddisfacibile in* $\mathcal{BAG}$.

Teorema 3 (Terminazione). *La procedura $SAT_{\mathcal{B}ag}(C)$ termina in un numero finito di passi. Inoltre il vincolo che ne risulta può essere* `false` *oppure un vincolo in forma risolta.*

Teorema 4 (Completezza). *Siano $C_1, \ldots, C_k$ i vincoli restituiti dalla procedura $SAT_{\mathcal{B}ag}(C)$, siano $N_i = FV(C_i) \backslash FV(C)$ ovvero le variabili libere. Allora vale:*

$$\mathcal{BAG} \models \forall (C \leftrightarrow \bigvee_{i=1}^{k} \exists N_i C_i)$$

Esempio 3 *Consideriamo ancora il vincolo*
$C \equiv a \in X \wedge B \in X \wedge X = \{\!\!\{ Y | Z \}\!\!\} \wedge B \neq a$ *dell'esempio 2. Sappiamo già che è soddisfacibile in $\mathcal{BAG}$; ora osserviamo che la procedura $SAT_{\mathcal{B}ag}(C)$ termina con successo e restituisce il seguente vincolo in forma risolta:*

$$X = \{\!\!\{ a, Z \}\!\!\} \wedge B = Z \wedge Y = a \wedge Z \neq a.$$

Per il secondo vincolo sempre dell'esempio 2, che non era sodisfacibile, la procedura $SAT_{\mathcal{B}ag}(C)$ restituisce `false`. □

2.4 Procedure di riscrittura dei vincoli

In questo paragrafo descriveremo le regole che vengono usate da $SAT_{\mathcal{B}ag}(C)$ per riscrivere un vincolo. Il procedimento avviene sempre nel seguente modo:

La procedura prende in input un vincolo C e ripetutamente sceglie un vincolo atomico c non ancora in forma risolta e vi applica le regole di riscrittura appropriate. La procedura si arresta quando C è in forma risolta oppure contiene un `false`*, come risultato della trasformazione di qualche c. Il procedimento avviene in modo non-deterministico dato che le regole di riscrittura possono comportare scelte non-deterministiche.*

while C	contains an atomic constraint c of the form $l\pi r$ not in solved form and $c \neq$`false` **do**
	select c;
	if $c =$ `false` **then return** `false`
	else if $c =$ `true` **then** erase c
	else apply to c any rewriting rule for $\mathcal{MS}$et-constraints della forma $l\pi r$;
return C.	

Figura 2.2: Ciclo fondamentale delle procedure di riscrittura dei vincoli

2.4.1 Vincolo di uguaglianza

L' algoritmo per verificare la soddisfacibilità e produrre le soluzioni a vincoli di uguaglianza per i multi-insiemi si chiama `Unify_msets` oppure algoritmo di unificazione tra multi-insiemi [4].

Esso produce in output `false` se il vincolo non è soddisfacibile, altrimenti fornisce una collezione di vincoli atomici di uguaglianza in forma risolta:

(1)	$X = X$	$\mapsto$	true
(2)	$X = t$ X *non è una variabile*	$\mapsto$	$X = t$
(3)	$X = t$ X *non occorre in* t X *occorre in* C	$\mapsto$	$X = t$ *e applico la sostituzione* X/t *a* C
(4)	$X = t$ $X \neq t,\ X$ *occorre in* t	$\mapsto$	false
(5)	$f(s_1, \ldots, s_m) = g(t_1, \ldots, t_m)$ $f \neq g$	$\mapsto$	false
(6)	$f(s_1, \ldots, s_m) = g(t_1, \ldots, t_m)$ $m \geq 0, \quad f \neq \{\!\{\cdot \mid \cdot\}\!\}$	$\mapsto$	$s_1 = t_1 \wedge \ldots \wedge s_m = t_m$
(7)	$\{\!\{t \mid s\}\!\} = \{\!\{t' \mid s'\}\!\}$ **tail**(s) *e* **tail**(s') *non sono la stessa variabile*	$\mapsto$	$(i) \quad (t = t' \wedge s = s') \vee$ $(ii) \quad (s = \{\!\{t' \mid N\}\!\} \wedge \{\!\{t \mid N\}\!\} = s')$
(8)	$\{\!\{t \mid s\}\!\} = \{\!\{t' \mid s'\}\!\}$ **tail**(s) *e* **tail**(s') *sono la stessa variabile*	$\mapsto$	**untail**$(\{\!\{t \mid s\}\!\}) =$ **untail**$(\{\!\{t' \mid s'\}\!\})$

Figura 2.3: Regole di riscrittura per l'uguaglianza di multi-insiemi.

Questo algoritmo utilizza due funzioni ausiliarie: **tail** e **untail** così definite:

$$\begin{cases} \mathbf{tail}(\varnothing) = \varnothing \\ \mathbf{tail}(X) = X & X \textit{ variabile} \\ \mathbf{tail}(\{\!\!\{ t \mid s \}\!\!\}) = \mathbf{tail}(s) \end{cases}$$

$$\begin{cases} \mathbf{untail}(X) = \varnothing & X \textit{ variabile} \\ \mathbf{untail}(\{\!\!\{ t \mid s \}\!\!\}) = \{\!\!\{ t \mid \mathbf{untail}(s) \}\!\!\} \end{cases}$$

La funzione **tail** restituisce il più interno fra gli elementi del secondo argomento del simbolo funzionale $\{\!\!\{ \cdot \mid \cdot \}\!\!\}$, tale termine viene detto *coda* del termine $\{\!\!\{ s \mid t \}\!\!\}$. Esso può essere il vuoto o una variabile.

Esempio 4 *Sia dato il multi-insieme* $\{\!\!\{ a, b, c \mid X \}\!\!\}$ *con* X *variabile non inizializzata; applicandovi la funzione tail si ottiene:*

$$\mathbf{tail}(\{\!\!\{ a, b, c \mid X \}\!\!\}) = X;$$

invece dato un multi-insieme completamente inizializzato si ottiene:

$$\mathbf{tail}(\{\!\!\{ a, b, c \}\!\!\}) = \varnothing.$$

□

Esempio 5 *Sia dato lo stesso multi-insieme dell'esempio precedente:* $\{\!\!\{ a, b, c \mid X \}\!\!\}$ *con* X *variabile non inizializzata; applicandovi la funzione untail si ottiene:*

$$\mathbf{untail}(\{\!\!\{ a, b, c \mid X \}\!\!\}) = \{\!\!\{ a, b, c \}\!\!\};$$

invece se consideriamo un multi-insieme completamente specificato abbiamo:

$$\mathbf{untail}(\{\!\!\{ a, b, c \}\!\!\}) = \{\!\!\{ a, b, c \}\!\!\}.$$

□

La **tail** e la **untail** vengono utilizzate dall'algoritmo di unificazione. Tale algoritmo distingue il caso di unificazione di due multi-insiemi aventi come coda la stessa variabile dal caso in cui le due code siano diverse, condizione che viene verificata attraverso la funzione **tail**. Nel primo caso si ha la regola di riscrittura apposita definita al punto (8), che utilizza la funzione **untail**.

2.4.2 Vincolo di disuguaglianza

Anche l'algoritmo di disuguaglianza usa le funzioni **tail** e **untail** precedentemente descritte.

E' necessario fornire alcune spiegazioni per alcuni passaggi di questo algoritmo, in particolare il caso (6.2), che tratta la situazione in cui i due multi-insiemi in oggetto non hanno la stessa coda variabile; infatti se usassimo l'assioma (E_k^m) otterremmo:

$$\{\!\{ t_1 \mid s_1 \}\!\} \neq \{\!\{ t_2 \mid s_2 \}\!\} \leftrightarrow \begin{array}{l} (t_1 \neq t_2 \vee s_1 \neq s_2) \wedge \\ \forall N (s_2 \neq \{\!\{ t_2 \mid N \}\!\} \vee s_1 \neq \{\!\{ t_1 \mid N \}\!\}) \end{array}$$

Dal momento che viene introdotto un quantificatore universale, questo è un vincolo più complesso di quelli atomici che costituiscono un vincolo della teoria $\mathcal{MS}et$. In alternativa potremmo riscrivere la regola usando la nozione intuitiva di appartenenza multipla: $x \in_i y$ significa che x appartiene al multiset y almeno i volte:

$$\{\!\{ t_1 \mid s_1 \}\!\} \neq \{\!\{ t_2 \mid s_2 \}\!\} \leftrightarrow \begin{array}{l} \exists X \exists n (n \in N \wedge \\ (X \in_n \{\!\{ t_1 \mid s_1] \}\!\} \wedge X \notin_n \{\!\{ t_2 \mid s_2 \}\!\}) \vee \\ (X \in_n \{\!\{ t_2 \mid s_2 \}\!\} \wedge X \notin_n \{\!\{ t_1 \mid s_1 \}\!\})) \end{array}$$

In questo caso notiamo che due multi-insiemi si differenziano per il nu-

mero di occorrenze dello stesso elemento, cioè ritroviamo una della proprietà fondamentali dei multi-insiemi.

Riportiamo di seguito le regole di riscrittura del vincolo di disuguaglianza nei multi-insiemi:

(1)	$d \neq d$ d *è una costante*	$\mapsto$	false
(2)	$f(s_1, \ldots, s_m) \neq g(t_1, \ldots, t_n)$ f *non è* g	$\mapsto$	true
(3)	$t \neq X$ t *non è una variabile*	$\mapsto$	$X \neq t$
(4)	$X \neq X$ X *è una variabile*	$\mapsto$	false
(5)	$f(s_1, \ldots, s_m) \neq g(t_1, \ldots, t_n)$ $n \geq 0, \quad f \neq cons_{\mathcal{MS}et}(\cdot, \cdot)$	$\mapsto$	$s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n$
(6.1)	$\{\!\{ t_1 \mid s_1 \}\!\} \neq \{\!\{ t_2 \mid s_2 \}\!\}$ **tail**(s_1) *e* **tail**(s_2) *sono la stessa variabile*	$\mapsto$	**untail**$(\{\!\{ t_1 \mid s_1 \}\!\}) \neq$ **untail**$(\{\!\{ t_2 \mid s_2 \}\!\})$
(6.2)	$\{\!\{ t_1 \mid s_1 \}\!\} \neq \{\!\{ t_2 \mid s_2 \}\!\}$ **tail**(s_1) *e* **tail**(s_2) *non sono la stessa variabile*	$\mapsto$	$(t_1 \neq t_2 \wedge t_1 \notin s_2) \vee$ (i) $(\{\!\{ t_2 \mid s_2 \}\!\} = \{\!\{ t_1 \mid N \}\!\} \wedge s_1 \neq N)$ (ii)
(7)	$X \neq f(t_1, \ldots, t_n)$ $X \in FV(t_1, \ldots, t_n)$	$\mapsto$	true

Figura 2.4: Regole di riscrittura per la disuguaglianza di multi-insiemi.

2.4.3 Vincolo di appartenenza e non appartenenza

Le regole di riscrittura per il vincolo di appartenenza in presenza di multi-insiemi si basano, come quelle per l'appartenenza, sulla definizione stessa di multi-insieme.

Riportiamo di seguito le tabelle che descrivono le regole di riscrittura sia dei vincoli di appartenenza semplice, che di non appartenenza nel caso di multi-insiemi

(1)	$r \in f(t_1, \ldots, t_n)$ $f \not\equiv \{\!\{\cdot \mid \cdot\}\!\}$	$\mapsto$	false
(2)	$r \in \{\!\{t \mid s\}\!\}$	$\mapsto$	$r = t \vee \quad (a)$ $r \in s \quad (b)$
(3)	$r \in X$ X *non occorre in* r	$\mapsto$	false
(4)	$r \in X$	$\mapsto$	$X = \{\!\{r \mid N\}\!\}$

Figura 2.5: Regole di riscrittura per l'appartenenza semplice nel caso di multi-insiemi.

(1)	$r \notin f(t_1, \ldots, t_n)$ $f \not\equiv \{\!\{\cdot \mid \cdot\}\!\}$	$\mapsto$	true
(2)	$r \notin \{\!\{t \mid s\}\!\}$	$\mapsto$	$r \neq t \wedge r \notin s$
(3)	$r \notin X$ X *occorre in* r	$\mapsto$	true

Figura 2.6: Regole di riscrittura per la non appartenenza nel caso di multi-insiemi.

2.5 Insiemi e multi-insiemi in CLP($\mathcal{SET}$+$\mathcal{BAG}$)

La presenza di un constraint solver in grado di verificare la soddisfacibilità dei vincoli su multi-insiemi permette di definire un linguaggio CLP capace di trattare con i vincoli su multi-insiemi. Questo linguaggio si ottiene dallo schema generale dei linguaggi CLP, istanziandolo sulla teoria e la struttura dei multi-insiemi, presentati nei paragrafi precedenti.

Il linguaggio che ne risulta, chiamato *CPL($\mathcal{BAG}$)*, viene visto come un'estensione naturale dei tradizionali linguaggi di programmazione logica, come ad esempio il PROLOG.

La sintassi di un programma *CPL($\mathcal{BAG}$)*, così come la semantica operazionale, logica e algebrica, sono le stesse dello schema generale dei linguaggi CLP.

Un *programma* P in *CPL($\mathcal{BAG}$)* è un insieme finito di regole del tipo:

$$A \ :- \ c, \overline{B}$$

dove A è una formula atomica (chiamata *head* della regola), c è un vincolo di $\mathcal{L}_{\mathcal{MS}et}$ e $\overline{B}$ è una congiunzione anche vuota, di formule atomiche. Un *goal* è una regola in cui *head* è vuota.

Esempio 6 (*Programma CLP($\mathcal{BAG}$)*). *Quello che segue è un esempio semplicissimo di programma in CLP($\mathcal{BAG}$), che definisce un predicato* `once(X,M)` *che risulta true se X occorre esattamente una volta nel multi-insieme M:*

```
once(X, {[ X | R]} ) :-
     X ∉ R.
```
□

La teoria sui multi-insiemi descritta nei paragrafi precedenti può essere combinata con la teoria sugli insiemi, generando così un nuovo linguaggio *CLP(*$\mathcal{SET} + \mathcal{BAG}$*)*, come suggerito in [3]. Questo è possibile dato che insiemi e multi-insiemi sono definiti allo stesso modo: entrambi sono regolati dall'assioma (W), e usano elementi dello stesso tipo. Inoltre ogni assioma all'interno di queste due teorie utilizza un solo simbolo di funzione binaria per rappresentare il costruttore della struttura stessa; $\{\cdot|\cdot\}$, per gli insiemi e $\{\!|\cdot|\cdot|\!\}$ per i multi-insiemi. In questo modo la teoria di una singola struttura dati non è influenzata dagli assiomi che regolano un'altra teoria.

La teoria assiomatica che combina insiemi e multi-insiemi si ottiene semplicemente dall'unione degli assiomi relativi ad entrambe queste strutture dati; analogamente la teoria di equivalenza combinata deriva dall'unione delle teorie di equivalenza $E_{\mathcal{MS}et}$ e $E_{\mathcal{S}et}$.

Per quanto riguarda il meccanismo di constraint solving, la procedura di riscrittura dei vincoli su insiemi viene combinata con quella relativa ai multi-insiemi per poter ottenere un constraint solver combinato. Tutte le regole di riscrittura dei vincoli di uguaglianza, disuguaglianza, appartenenza semplice e non appartenenza su multi-insiemi, possono essere ottenute in modo parametrico da quelle relative agli insiemi, sostituendo semplicemente il simbolo di costruttore della struttura.

La procedura $SAT_{\mathcal{B}ag}$ che verifica la soddisfacibilità di un vincolo nella teoria $\mathcal{MS}et$, può essere sostituita nella teoria combinata, da una generica procedura SAT in cui ogni chiamata alle regole di riscrittura dei vincoli relativi ai multi-insiemi, viene sostituita dalla chiamata alla composizione di: $Unify\_bags \cup Unify\_set(neq\_bag \cup neq\_set(nin\_bag \cup nin\_set(in\_bag \cup$

in_set))).

Dato che i teoremi 2, 3 e 4 garantiscono la soddisfacibilità in $\mathcal{BAG}$ dei vincoli in forma risolta del linguaggio relativo $\mathcal{L}_{\mathcal{B}ag}$, la terminazione e la completezza della procedura $SAT_{\mathcal{B}ag}$, e questo vale rispettivamente anche per la teoria relativa agli insiemi, allora essi valgono anche per la procedura SAT relativa alla teoria combinata.

Il linguaggio ottenuto dalla combinazione di queste due teorie permette di mescolare liberamente termini di natura diversa: insiemi e multi-insiemi.

Capitolo 3

Risoluzione di vincoli in JSetL

3.1 Introduzione a JSetL

JSetL è una libreria Java che aggiunge a Java alcune funzionalità per supportare la progammazione dichiarativa; tra queste, variabili logiche, unificazione, strutture dati ricorsive (liste ed insiemi), non-determinismo e constraint solving, tipiche dei linguaggi CLP (Constraint Logic Progamming). JSetL fornisce molte utilità tipiche di CLP(SET), un linguaggio di programmazione logica a vincoli con gli insiemi. Infatti JSetL permette al programmatore di creare insiemi (anche parzialmente specificati) e di manipolarli attraverso i vincoli. Gli insiemi forniscono un modo naturale per esprimere il non-determinismo e per raggruppare tutte le soluzioni dei problemi che ne ammettono più di una.

JSetL possiede tutte queste caratteristiche; in questa libreria i vincoli e il processo per la loro risoluzione sono solo uno degli strumenti usati per supportare la programmazione dichiarativa. Uno degli scopi di JSetL è unire le caratteristiche della DP con quelle del linguaggio Java; d'altra parte l'efficienza nella computazione non è un requisito primario per JSetL.

3.2 Variabili logiche e strutture dati di JSetL

Ci preoccupiamo ora di dare una rapida occhiata alla libreria JSetL e alle sue principali caratteristiche per poi affrontare il discorso del constraint solving. I linguaggi logici come il Prolog, e alcuni linguaggi imperativi per la programmazione con vincoli come ad esempio Alma-0, permettono di utilizzare variabili logiche che non devono necessariamente avere un valore al momento della loro dichiarazione. L'accesso a queste variabili è possibile anche quando non hanno alcun valore e non genera nessun errore. Inoltre, contrariamente a quanto accade per le variabili dei linguaggi imperativi, dopo che ad una variabile logica è stato assegnato un valore, questo non può essere modificato da un altro assegnamento.

Nella definizione della libreria JSetL è prevista l'esistenza di oggetti che hanno le stesse caratteristiche di una variabile logica. Tali oggetti sono realizzati come istanze della classe `Lvar`. JSetL fornisce inoltre due nuovi tipi di strutture dati: insiemi e liste. Oggetti di questo tipo sono realizzati come istanze delle classi `Set` e `Lst` rispettivamente.

Alle variabili logiche può essere assegnato un valore oppure no, cioè possono essere *inizializzate* oppure *non inizializzate*. Il valore di una variabile logica in JSetL può essere di qualsiasi tipo. Un valore può essere assegnato ad una `Lvar` non inizializzata come risultato del processo di constraint solving che coinvolge tale variabile, in particolare attraverso il vincolo di uguaglianza. I vincoli vengono usati sia per inizializzare che per controllare il valore di una variabile logica, ma non possono essere usati per modificare tale valore.

Una *variabile logica* è un'istanza della classe `Lvar`. Una variabile logica con il nome `VarName` può essere creata attraverso la seguente *dichiarazione Java di una variabile logica*:

```
Lvar VarName = new Lvar(LvarNameExt,VarValue);
```

dove `VarNameExt` è un nome esterno (parametro opzionale), mentre `VarValue` è un valore (anch'esso opzionale) di inizializzazione della variabile stessa.

Le strutture dati `Set` e `Lst` vengono trattate come le variabili logiche, e JSetL fornisce, anche per queste strutture dati, metodi per costruire e manipolare oggetti di tipo `Set` e `Lst`.

La principale differenza tra insiemi e liste risiede nell'importanza attribuita all'ordine e alle ripetizioni degli elementi nelle liste, aspetti irrilevanti negli insiemi. Entrambe le strutture, eventualmente vuote, sono costituite da una sequenza di elementi che possono essere in numero arbitrario e di tipo diverso. Sia gli insiemi che le liste possono essere parzialmente specificate, per esempio essi possono contenere variabili logiche non ancora inizializzate. In questo caso la cardinalità dell'insieme può variare in base al valore assunto dalle variabili che vi compaiono.

Analogamente alle `Lvar` anche per i `Set` e le `Lst` esistono in JSetL le relative classi che permettono di creare oggetti di questo tipo:

```
Set SetName = new Set(SetNameExt,SetElemValues);
Lst LstName = new Lst(LstNameExt,LstElemValues);
```

dove `SetNameExt\LstNameExt` è il solito parametro opzionale, rappresentante il nome esterno dell'insieme o della lista che si sta creando, mentre `SetElemValues\LstElemValues` è un valore anch'esso opzionale di inizializ-

zazione dell'insieme\lista.

Un insieme, così come una lista, può essere creato vuoto (attraverso la costante `Set.empty`\`Lst.empty`), inizializzato o non inizializzato.

Il valore di un insieme è la collezione dei suoi elementi. In particolare, come succede anche per le liste, quando un oggetto di tipo `Set`\`List` viene creato attraverso l'operatore **new**, il suo valore può essere specificato nei seguenti modi:

- passando un array di elementi di tipo *t* come parametro del costruttore, sia di una lista che di un insieme, ad esempio:

```
t[] elem = {t1,t2,t3};
Lst l = new Lst(LstNameExt,elem);
Set s = new Set(SetnameExt,elem);
```

- passando i limiti destro e sinistro *l* e *u* di un intervallo *[l,u]* di numeri interi, dove se $u<l$ la struttura creata sarà vuota

```
Lst l2 = new Lst(LstnameExt,l,u);
Set s2 = new Set(SetnameExt,l,u);
```

- passando un oggetto di tipo *Lst* o *Set* rispettivamente,

```
Lst l3 = new Lst();
Set s3 = new Set();
Lst l4 = new Lst(LstNameExt,l3);
Set s4 = new Set(SetNameExt,s3);
```

Sia la classe `Set` che la classe `Lst` permettono di costruire strutture *non limitate*, cioè con un certo numero di elementi noti, che possono eventualmente essere `Lvar` o anche liste o insiemi, e un *"resto"* dato da un `Set` o da una `Lst` non inizializzate rispettivamente. Per costruire `Set` e `Lst` non limitate bisogna utilizzare le funzioni `ins` e `insAll` definite in queste classi.

3.3 Programmare con i vincoli

3.3.1 Vincoli in JSetL

Definizione 9 *(***Vincolo atomico***) In JSetL un* **vincolo atomico** *è una relazione binaria o ternaria, definita mediante le espressioni:*

- $e_1.op(e_2);$
- $e_1.op(e_2,e_3);$

dove **op** *è uno dei seguenti metodi* **(eq, neq, in, nin, subset, union, inters, disj, differ, nunion, ninters, nsubset, ndisj, ndiffer, less, le, ge, gt)** *e* e_1**,** e_2 *ed* e_3 *sono espressioni il cui tipo dipende da op.*

Definizione 10 *(***Vincolo***) Un* **vincolo** *(constraint) 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'esso un vincolo.*

Sia l_1 un'espressione di tipo `Lvar, Lst` o `Set`, l_2 un'espressione di tipo primitivo, o di tipo `String, Lvar, Lst` o `Set`, m_1 una `Lvar` inizializzata con

un intero, e sia m_2 una `Lvar` inizializzata con un intero o con un'espressione di tipo `int` o `Integer`. Infine, siano s_1, s_2 e s_3 espressioni di tipo `Set`. Di seguito riportiamo i vincoli definiti in JSetL:

- **Uguaglianza e disuguaglianza**: *l_1.eq(l_2), l_1.neq(l_2)*
- **Appartenenza e non appartenenza** ad un insieme: *l_1.in(l_2), l_1.nin(l_2)*
- Vincoli di **confronto**: *m_1.le(m_2), m_1.lt(m_2), m_1.ge(m_2), m_1.gt(m_2)*
- Vincoli **aritmetici**: *m_1.sum(m_2), m_1.sub(m_2), m_1.mul(m_2), m_1.div(m_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 e non inclusione**: *s_1.subset(s_2), s_1.nsubset(s_2)*
- **Intersezione e non intersezione**: *s_1.inters(s_2), s_1.ninters(s_2)*
- **Differenza e non differenza**: *s_1.differ(s_2,s_3), s_1.ndiffer(s_2,s_3)*
- Vincolo **less**: *s_1.less(s_2,s_3)*

A questi vanno aggiunti i metodi `allDifferent` e `forall`, che anche se di fatto non sono vincoli, vengono aggiunti al constraint store, come se lo fossero.

3.3.2 Constraint Store

Il *Constraint Store* è un oggetto della classe `Store`. In esso vengono memorizzati tutti i vincoli attivi di un programma.

JSetL fornisce i metodi attraverso i quali è possibile aggiungere nuovi vincoli al *constraint store*, visualizzarne il contenuto e rimuovere tutti i vincoli

ivi presenti. L'introduzione di un nuovo vincolo avviene attraverso il metodo `add` della classe `SolverClass`, cioè invocando:

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

che aggiunge il vincolo `C` al constraint store di `S`, dove `S` è il risolutore dei vincoli. L'insieme di tutti i vincoli memorizzati nel constraint store viene interpretato come la congiunzione di vincoli atomici.

Alla fine, quando tutti i vincoli sono stati aggiunti al constraint store, possiamo risolverli invocando il metodo `solve` della classe `SolverClass`:

```
S.solve();
```

Questo metodo ricerca, in modo non-deterministico, una soluzione che soddisfi tutti i vincoli introdotti. Se non esiste alcuna soluzione, viene generato un `Failure` e la computazione termina.

Il metodo `solve` tiene traccia delle alternative rimaste inesplorate durante la ricerca e in caso di backtracking torna ad un punto di scelta precedente che abbia ancora delle alternative aperte e continua la ricerca da quel punto fino a trovare, se esiste, una soluzione.

3.3.3 Constraint solving in JSetL

L'approccio adottato per constraint solving in JSetL è lo stesso di CLP($\mathcal{SET}$). Il risolutore cerca di ridurre ciascun vincolo presente nel constraint store in una forma semplificata, detta *forma risolta*, che può essere testata facilmente per determinare se è 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.

Definizione 11 *(***Forma Risolta***) Un vincolo (vincolo) è in* **forma risolta** *se è vuoto, oppure se tutti i vincoli che lo compongono sono in una delle seguenti forme:*

1. *x.**eq**(e), x non è contenuta nè in e, nè negli altri vincoli di C;*
2. *x.**neq**(e), non c'è nessuna occorrenza di x in e;*
3. *o.**nin**(x), non c'è nessuna occorrenza di x in e;*
4. *x_1.**disj**(x_2), dove $x_1 \neq (x_2)$;*
5. *$x_3 = x_1$.**union**(x_2), con $x_1 \not\equiv x_2$ e non ci sono disequazioni della forma x_i.**neq**(t) o t.**neq**(x_i) in C per ogni $i = 1, 2, 3$;*

Un vincolo in forma risolta ottenuto da un vincolo C rappresenta una soluzione per C. Un vincolo può avere soluzioni multiple. Il risolutore di vincoli in JSetL è in grado di trovare in modo non-deterministico tutte le soluzioni corrette per un dato vincolo. Il non-determinismo si ottiene attraverso *punti di scelta* ovvero *choice points* e *backtracking*. Se il processo di riduzione dei vincoli porta ad un fallimento, la computazione torna indietro al punto di scelta aperto, più recente (backtracking cronologico). Se non esistono punti di scelta ancora aperti, il processo di riduzione dei vincoli fallisce e viene generata un'eccezione di tipo `Failure`.

Definizione 12 *(***Failure***) Diciamo che una computazione termina con un fallimento se essa lancia l'eccezione* **Failure**. *Altrimenti, la computazione termina con successo.*

I vincoli in forma risolta sono *irriducibili*, e vengono lasciati nel constraint store, senza essere influenzati da nuove invocazioni del metodo `solve`. Una computazione che termina con successo, vedrà il constraint store contenente

un insieme non vuoto di vincoli in forma risolta contrariamente, i vincoli che non sono in forma risolta, vengono eliminati durante il processo di constraint solving: infatti, essi vengono ridotti in vincoli in forma risolta, o vengono riscritti come `false`, oppure possono lanciare un'eccezione.

La versione attuale di JSetL fornisce un constraint solver che non è in grado di trattare in modo completo con vincoli con interi; ovvero vincoli interi che contengono variabili logiche non inizializzate, vengono tralasciati il più possibile, lasciandoli irrisolti nel constraint store. Nel momento in cui bisogna controllare la soddisfacibilità dei vincoli, se nel constraint store sono presenti vincoli interi con variabili logiche non inizializzate, viene lanciata l'eccezione

`Uninitialized_Variable_in_arithmetical_expression.`

In altre parole, in questi casi particolari, il constraint solver informa che non ci sono sufficienti informazioni per decidere la soddisfacibilità dei vincoli dati.

Il constraint solver di JSetL è un'implementazione Java del solver di CLP($\mathcal{SET}$), esteso con semplici vincoli con gli interi. Il constraint solver di JSetL è implementato attraverso la classe `SolverClass` e il metodo principale per il processo di constraint solving è `solve`. Dato `Solver`, istanza della classe `SolverClass`, l'invocazione di `Solver.solve()`, permette di scoprire se i vincoli nel constraint store non sono soddisfacibili, e in tal caso lanciare l'eccezione `Failure`, oppure trasformarli in modo non-deterministico, in una forma semplificata.

Esempio 7 . *Semplice esempio di risoluzione di vincoli in JSetL. Controllare l'appartenenza di un elemento* x *alla differenza tra i due insiemi* s_1,s_2, *ovvero* $(x \in s_1 \backslash s_2)$.

```
public static void in_difference(Lvar s, Set s1, Set s2)
throws Failure{
     Solver.add(x.in(s1));
     Solver.add(x.nin(s2));
     Solver.solve();
     }
```

Supponendo di eseguire il seguente frammento di codice in un `main`

```
in_difference(x,s,r);
x.output();
Solver.showStore();
```

dove `s` e `r` sono rispettivamente gli insiemi {`1,2`} e {`1,3`}, e `x` è una variabile logica non inizializzata. L'output generato da questo programma è il seguente:

```
x = 2
Store: empty
```

Contrariamente, se `x` è un insieme non inizializzato, eseguendo lo stesso frammento di programma, avremmo il seguente output:

```
x = unknown
s = {x|Set_1}
Store: x.neq(1) x.neq(3)
```

che viene interpretato nel seguente modo: `s` può essere un qualunque insieme contenente l'elemento `x` e `x` deve essere diverso sia da `1` che da `3`.

□

3.4 Non-determinismo

In JSetL il non-determinismo è utilizzabile soltanto all'interno della definizione dei vincoli, siano essi predefiniti come `eq, neq, in, nin, ...` oppure definiti dall'utente. Il non-determinismo è stato realizzato introducendo i punti di scelta ed il backtracking.

Se un vincolo v ammette n soluzioni, sol_0,...,sol_{n-1}, allora il metodo che risolve questo tipo di vincolo tiene traccia delle alternative rimaste inesplorate. In questo modo, se in seguito vengono introdotti nuovi vincoli e questi non sono soddisfatti dalla soluzione trovata in precedenza, potranno essere esplorate altre alternative per cercare una soluzione che soddisfi tutti i vincoli. Per capire meglio come è possibile ottenere soluzioni in modo non-deterministico, riportiamo un esempio in cui vengono risolti vincoli non-deterministici e viene applicato il backtracking, mentre per le spiegazioni riguardanti gli aspetti implementativi riguardanti il non-determinismo e più in generale la libreria JSetL, rimandiamo l'attenzione al paragrafo 3.5.1.

Esempio 8

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

   SolverClass Solver = new SolverClass();

   int[] ar = {1,1,3,2};
   MultiSet m = new MultiSet("m", ar);

   int[] br = {1,3,3};
   MultiSet n = new MultiSet("n", br);
```

```
    Lvar X = new Lvar("X");
    Lvar Y = new Lvar("Y");

    MultiSet s = new MultiSet("s", MultiSet.empty.ins(Y).ins(X));

    Solver.add(Y.in(m));    //vincolo di appartenenza
    Solver.add(X.in(n));    //vincolo di appartenenza
    Solver.add(Y.nin(n));   //vincolo di non appartenenza

    Solver.solve();
    s.output();
    return;
     }
}
```

Il problema consiste nel determinare una soluzione per i vincoli $Y \in \{\!\{ 1, 1, 3, 2 \}\!\}$ $\wedge\, X \in \{\!\{ 1, 3, 3 \}\!\} \wedge Y \notin \{\!\{ 1, 3, 3 \}\!\}$ con X, Y variabili logiche non inizializzate. Inizialmente lo stack delle alternative risulta vuoto e questi vincoli popolano lo store. La risoluzione del vincolo $Y \in \{\!\{ 1, 1, 3, 2 \}\!\}$ comporta una scelta non-deterministica, in quanto è soddisfatto sia con $Y = 1$ per due volte, sia con $Y = 3$ che con $Y = 2$. Quindi il risolutore procede scegliendo una delle alternative e creando 3 punti di scelta per le restanti alternative che vengono introdotti nello stack: A_1, A_2 e A_3. Quindi nello store il primo vincolo è stato risolto come $Y = 1$, poi viene analizzato il secondo vincolo $X \in \{\!\{ 1, 3, 3 \}\!\}$ che comporta un'altra scelta non-deterministica in quanto ammette diverse soluzioni: una soluzione è data da $X = 1$ e un'altra, considerata due volte, è $X = 3$. A questo punto nello stack delle alternative sono state aggiunte anche A_4, A_5. La risoluzione dell'ultimo vincolo $Y \notin \{\!\{ 1, 3, 3 \}\!\}$ porta ad un fallimento in quanto $Y = 1 \wedge Y \notin \{\!\{ 1, 3, 3 \}\!\}$. Quindi viene applicato il back-tracking, cioè viene estratto un punto di scelta dallo stack e viene esplorata

un'altra alternativa. Nel nostro caso viene estratta l'alternativa A_5, che era stata generata durante la risoluzione del vincolo $X \in \{\!\!\{1, 3, 3\}\!\!\}$. In questo modo l'alternativa A_5 viene eliminata dallo stack e lo store si presenta nel seguente modo: $Y = 1 \wedge X = 3 \wedge Y \notin \{\!\!\{1, 3, 3\}\!\!\}$, ma è ancora inconsistente. Viene applicato il backtraking allo stesso modo fino ad arrivare alla seguente situazione: nello stack sono ancora aperte le alternative A_1 e A_2. Viene aperta la A_3 e lo store e le variabili vengono riportate nelle condizioni in cui erano quando è stato creato tal punto di scelta, cioè: $Y = 2, X = 1$ e $Y \notin \{\!\!\{1, 3, 3\}\!\!\}$. Quindi la soluzione considerata è $s = \{\!\!\{1, 2\}\!\!\}$ e la computazione termina. Nello stack restano memorizzate due punti di scelta, in quanto non sono state esplorate tutte le alternative, infatti volendo ottenerle tutte troveremmo anche $s = \{\!\!\{3, 2\}\!\!\}$ per due volte.

□

3.5 Aspetti implementativi di JSetL

3.5.1 Vincoli e constraint store

Nei paragrafi precedenti abbiamo parlato di constraint store e di risoluzione dei vincoli in modo non-deterministico. Le classi della libreria JSetL coinvolte sono `SolverClass`, `Constraint`, `ChoicePoint`, `Backtracking`, `VarState` e `StoreState`.

In JSetL un singolo vincolo è un oggetto della classe `Constraint`, mentre una congiunzione di vincoli è un oggetto di tipo `ConstraintsConjunction`, cioè un vettore i cui elementi sono i vincoli della congiunzione.

Di seguito riportiamo gli attributi della classe `Constraint`:

```
protected Object arg1;
protected Object arg2;
protected Object arg3;
protected Object arg4 = null;
protected int cons;
protected int caseControl = 0;
```

Ogni istanza della classe ha 4 campi di tipo `Object` in cui vengono memorizzati gli elementi coinvolti nel vincolo. Nel caso di vincoli binari vengono usati solo due di questi attributi. La classe possiede poi altri due campi: l'attributo intero `cons` in cui viene memorizzato il tipo di vincolo, e l'attributo `caseControl` che di default è uguale a 0. Quest'ultimo viene utilizzato nel caso in cui la risoluzione del vincolo richieda una scelta non-deterministica. La classe `ConstraintsConjunction` estende la classe `Vector`. Utilizzando il metodo `and` di questa classe è possibile definire vincoli ottenuti come congiunzione di altri vincoli.

Il metodo `and` della classe `ConstraintsConjunction` è definito come segue:

```
public ConstraintsConjunction and(Constraint s) {
super.add(s);
return this;
}
```

I metodi che definiscono i vincoli non fanno altro che creare e restituire oggetti di tipo `Constraint`. La classe `Solver` si occupa dell'introduzione dei vincoli nel constraint store e della loro risoluzione.

Gli attributi della classe `SolverClass` sono:

```
protected boolean storeInvariato = true;
protected int storeSize = 0;
protected ConstraintStore store = new ConstraintStore(this);
protected RewritingConstraintsRules C =
                        new RewritingConstraintsRules(this);
protected Backtracking   B  = new Backtracking(this);
protected StoreState     SS = new StoreState(this);
```

Questi attributi sono accessibili solo dalla classe `SolverClass` stessa.

Il constraint store è realizzato mediante l'attributo `store` della classe `ConstraintStore` che estende la classe `ConstraintsConjunction`, i cui elementi sono riferimenti a istanze della classe `Constraint`. L'inserimento di un singolo vincolo, o di una congiunzione di vincoli, nello store è realizzato dal metodo `add` della classe `SolverClass`, che riportiamo di seguito:

```
public void add(Constraint s) {
  store.add(s);
  return;
  }

public void add(ConstraintsConjunction storeVector) {
      for(int i = 0; i < storeVector.size(); i++){
      store.add((Constraint)storeVector.get(i));
  }
  return;
}
```

Il metodo `add` è stato sovraccaricato: il parametro passato può essere un oggetto di tipo `Constraint`, cioè un vincolo singolo, oppure un oggetto di

tipo `ConstraintsConjunction`, cioè una congiunzione di vincoli. Nel primo caso il metodo aggiunge il vincolo passato come parametro in coda allo store, nel secondo caso aggiunge in coda tutti i vincoli della congiunzione.

3.5.2 Punti di scelta e backtracking

La risoluzione dei vincoli introdotti avviene mediante il metodo `solve` della classe `SolverClass`. Questo metodo cerca di trasformare ciascun vincolo in forma risolta, se questo processo ha successo significa che l'insieme dei vincoli introdotti ha almeno una soluzione. Come abbiamo visto nell'esempio 8 la riscrittura dei vincoli in forma risolta richiede una scelta non-deterministica tra diverse alternative. Il metodo `solve` si preoccupa di tenere traccia di tutte le alternative rimaste inesplorate. Se un vincolo ammette più di una soluzione allora il metodo che lo risolve avrà un'istruzione `switch` contenente `n` blocchi case: uno per ciascuna delle alternative. Il valore dell'espressione `switch` è dato dal valore dell'attributo `caseControl` del vincolo che si sta risolvendo. Questo attributo è `0` per default, in questo modo la prima alternativa che viene considerata è quella la cui etichetta `case` è uguale a `0`.
Ciascun blocco `case`, ad eccezione dell'ultimo, crea un punto di scelta e lo aggiunge allo stack delle alternative.

Un punto di scelta è un oggetto della classe `ChoicePoint`. Ciascuna istanza di questa classe ha i seguenti attributi:

```
int cont;
VarState   varsState;
StoreState storeState;
```

Gli attributi di questa classe sono: l'intero `cont` che permette di memorizzare il valore che deve essere assegnato al campo `caseControl` del vincolo che ha aperto l'alternativa, l'oggetto `varsState`, istanza della classe `VarState`,

che permette di memorizzare le variabili che al momento della creazione del punto di scelta non sono inizializzate e l'oggetto `storeState`, istanza della classe `StoreState`, che permette di memorizzare lo stato dello store e il vincolo che ha aperto l'alternativa. La classe `VarState` possiede i seguenti attributi:

```
Vector lvarNonIniz;
Vector setNonIniz;
Vector lstNonIniz;
```

Il primo è un vettore i cui elementi sono dei riferimenti a delle `Lvar`, gli elementi del secondo vettore sono dei riferimenti a delle `Lst` e gli elementi del terzo sono dei riferimenti a dei `Set`. La classe `StoreState` possiede i seguenti attributi:

```
ConstraintStore statoStore;
int posizione;
```

L'attributo `statoStore` serve per memorizzare un clone dello store, mentre l'attributo `posizione` serve per memorizzare la posizione nello store del vincolo che ha generato il punto di scelta. Per creare il punto di scelta e introdurlo nello stack delle alternative viene utilizzato il metodo `addChoicePoint` della classe `Backtracking`. Questo metodo è definito come segue:

```
protected void addChoicePoint(int i, VarState v, StoreState s) {
        ChoicePoint newCP = new ChoicePoint(i, v, s);
        this.alternatives.push(newCP);
        return;
}
```

Il metodo crea un oggetto `ChoicePoint` e lo memorizza nell'attributo `alternatives` di tipo `Stack` della classe `Backtracking` che è un membro protetto della classe `SolverClass`. Il parametro `v` è determinato dal metodo della classe `VarState cloneVarState`, il parametro `s` è invece determinato dal metodo statico della classe `StoreState cloneStoreState`.

L'oggetto restituito dal metodo `cloneVarState` è un'istanza della classe `VarState` i cui attributi, come abbiamo detto, sono tre vettori. Gli elementi di questi vettori sono dei riferimenti, rispettivamente, alle `Lvar`, alle `Lst` e ai `Set` non inizializzati nel momento in cui viene invocato il metodo. Il metodo `cloneStoreState`, invece, fa una copia dello store e memorizza in che posizione dello store si trova il vincolo che ha generato il punto di scelta, questo metodo infatti restituisce un oggetto di tipo `StoreState`.

Dopo aver creato e memorizzato il punto di scelta, il blocco `case` selezionato applica una delle possibili soluzioni per il vincolo. Se la soluzione considerata porta ad un fallimento, viene applicato il backtracking estraendo un punto di scelta dallo stack delle alternative. L'attributo `cont` dell'oggetto `ChoicePoint` estratto viene assegnato all'attributo `caseControl` del vincolo che ha generato quel punto di scelta e gli attributi `VarState` e `StoreState` vengono utilizzati per riportare le variabili e lo store allo stato in cui erano prima di esplorare l'alternativa che ha portato al fallimento.

3.5.3 Trattamento dei vincoli

Consideriamo la classe `Set`, in essa sono definiti tutti i vincoli che possono coinvolgere gli insiemi: vincoli di uguaglianza, disuguaglianza, appartenenza, non appartenenza, unione, intersezione ed altri. Ad esempio la funzione

```
public Constraint eq (Object z) {
Constraint s = new Constraint(this, Enviroment.eqCode, z);
```

```
    return s;
    }
```

definisce il vincolo di uguaglianza tra un insieme e un oggetto di classe `Object`. Essa restituisce un nuovo vincolo, cioè un oggetto della classe `Constraint`, in cui il primo argomento ($arg1$) viene inizializzato con il `Set` su cui è stato invocato il metodo, il secondo argomento ($arg2$) viene inizializzato con il parametro `z` che abbiamo passato alla funzione e l'attributo `cons` della classe `Constraint` prende il valore `Enviroment.eqCode`. Ad esempio, se inseriamo un nuovo vincolo nel constraint store, attraverso la funzione `add`, scrivendo $Solver.add(M.eq(N))$, dove M ed N sono `Set` e $Solver$ è un oggetto di tipo `SolverClass`, viene chiamata la funzione `eq` della classe `Set` che restituisce un oggetto di tipo `Constraint` che ha come primo argomento ($arg1$) il multi-insieme d'invocazione M, come secondo argomento ($arg2$) il parametro passato N e il codice del constraint, memorizzato nell'attributo $eqCode$ di un oggetto di tipo `Enviroment`.

Per risolvere tutti i vincoli memorizzati nel constraint store si usa la funzione `solve()` della classe `SolverClass`:

```
public void solve() throws Failure {
        if (store.existForall()) {
            store.getConstraintStoreVars();
            if(store.ctrlConstraintStoreVars())
                            throw new IllegalVariableUse();
        }
    if(B.alternatives.size() > 0) {
        B.updateStoreAlternative();
        B.updateVarAlternative();
    }
    do {
```

```
        try { ris(); }
        catch(Fail f){ B.gestisciFallimento(); }
    } while(!storeInvariato);

    storeSize = store.size();
    return;
    }
```

Il metodo `solve` scorre lo store e risolve i vincoli che non sono ancora in forma risolta. Il campo booleano `storeInvariato` permette di stabilire se il constraint store è già in forma risolta o se è necessario ricominciare a scorrerlo per continuare a risolvere i vincoli che non sono ancora in forma risolta. Il valore di `storeInvariato` è `true` quando una passata della `solve` non produce alcuna modifica nel constraint store. Se un vincolo è inconsistente viene sollevata un'eccezione `Fail` dal metodo `fail`; se c'è un fallimento il `catch` della `solve()` cattura l'eccezione e interrompe la `solve` passando il controllo al metodo `gestisciFallimento()` della classe `Backtracking` che innesca il backtracking (vedi paragrafo 3.5.1). I metodi `updateStoreAlternative()` e `updateVarAlternative()` permettono di aggiungere ai punti di scelta, creati da una `solve` precedente, i vincoli introdotti successivamente alla loro creazione. In questo modo in caso di backtracking non vengono persi i vincoli introdotti dopo la creazione del punto di scelta al quale il risolutore è tornato durante il backtracking. Quando la `solve` arriva al termine dello store, se lo store è rimasto invariato, termina altrimenti ricomincia la risoluzione dei vincoli.

Ogni vincolo nel constraint store viene risolto invocando la funzione `ris()` della classe `SolverClass`:

```
    private void ris() throws Failure {
```

```
        storeInvariato = true;
        int i = 0;
        do {
            C.risElem(i);
            i++;
        } while(i < store.size());
    return;
    }
```

la quale, a sua volta richiama la funzione `risElem` che risolve uno per uno tutti i vincoli. Essa analizza il parametro `cons` di ogni vincolo e richiama la funzione appropriata della classe `RewritingConstraintsRules` per risolverlo. Nel caso di vincolo di uguaglianza ad esempio, viene invocata:

```
protected void eq(Constraint s) throws Failure;
```

la quale, in base al tipo degli argomenti coinvolti nel vincolo, richiamerà il metodo che implementa le giuste regole di riscrittura del vincolo. Nel nostro caso, in presenza del vincolo $M.eq(N)$ si accede alla funzione:

```
protected void eqSet(Set set1, Set set2, Constraint s) throws
                                                        Failure.
```

Capitolo 4

Introduzione dei multi-insiemi in JSetL

In questo capitolo descriveremo le nuove possibilità che abbiamo aggiunto alla libreria JSetL per permettere la gestione di multi-insiemi.

4.1 Creazione di un multi-insieme

Definizione 13 *(***Multi-insieme***) Un* **multi-insieme** *è una collezione (anche vuota) di oggetti di tipo qualsiasi (gli elementi del multi-insieme). In JSetL un multi-insieme è un'istanza della classe* **MultiSet***, creata attraverso la dichiarazione Java*

```
MultiSet MSetName = new MultiSet(MSetNameExt,MSetElemValues);
```

dove **MSetName** *è il nome del multi-insieme,* **MSetNameExt** *è un nome opzionale per il multi-insieme, e* **MSetElemValues** *è un parametro anch'esso opzionale, usato per specificare gli elementi del multi-insieme.*
La costante **MultiSet.empty** *è usata per denotare il* **multi-insieme vuoto**.

I multi-insiemi vengono creati attraverso l'operatore **new**: elencando gli elementi che lo compongono esplicitamente oppure inserendoli in un array.

Gli elementi di un multi-insieme possono essere, in particolare, variabili logiche, insiemi, liste o multi-insiemi stessi, anche parzialmente inizializzati. I costruttori per la classe `MultiSet` sono i seguenti:

```
public MultiSet();
public MultiSet(String n);
public MultiSet(type[] arr);
public MultiSet(String n, type[] arr);
```

ovvero il costruttore di default, il costruttore con il solo parametro `String` in cui si vuole dare il nome `n` al multi-insieme che si sta creando, e due costruttori che prendono in input un array di elementi di tipo `type`, uno di essi prende come paramentro anche una stringa, che ha la stessa funzione di quella passata al costruttore con il solo parametro `String`. Il tipo `type` può essere `boolean, byte, short, int, long, float, double` e `char`, oppure di tipo `Object` che è una superclasse per tutti i tipi più complessi, come ad esempio `String, Lvar, Lst` e `MultiSet`, ma non solo.

Definizione 14 *(***Multi-insieme parzialmente specificato***) Un multi-insieme che contiene alcuni elementi che non sono inizializzati, è detto* **multi-insieme parzialmente specificato**.

Definizione 15 *(***Multi-insiemi annidati***) Gli elementi di un multi-insieme che sono loro stessi multi-insiemi, sono detti* **multi-insiemi annidati**. *L'annidamento può avvenire ad ogni livello.*

Vediamo alcuni esempi di come dichiarare ed inizializzare un multi-insieme:

Esempio 9

```
MultiSet M = new MultiSet();
MultiSet N = MultiSet.empty;
MultiSet R = new MultiSet("R");

int[] ar = {1,2,2};
MultiSet V = new MultiSet("V", ar);
```

Il primo, M, è un multinsieme non inizializzato e con nome non specificato dall'utente; quindi il parametro `MSetNameExt` della Definizione 13 viene sostituito dal nome di default per i multi-insiemi: `MultiSet_1`, dove il numero 1 indica che questo è il primo multi-insieme che abbiamo costruito. Diversamente accade per il multinsieme R, che risulta sempre non inizializzato, ma ha come nome R. Il secondo multi-insieme dell'esempio rappresenta la costante, di cui alla Definizione 13, che indica l'aggregato vuoto. Infine $V = \{\!\{ 1, 2, 2 \}\!\}$, cioè nell'insieme vuoto N vengono inseriti tutti gli elementi dell'array `ar`, ovvero abbiamo dichiarato un multi-insieme totalmente specificato. □

4.2 Multi-insieme creato come risultato di `ins` e `insAll`

Per inserire degli elementi in un multi-insieme, possiamo usare due funzioni diverse: `ins` e `insAll`, per aggiungere ad un `MultiSet` un singolo valore oppure tutti i valori di un array, rispettivamente.
Esse vengono invocate su un multi-insieme e restituiscono un multi-insieme

ottenuto da quello di invocazione aggiungendo gli elementi passati come parametro:

```
public MultiSet ins(Number i);
public MultiSet ins(Object o);
public MultiSet insAll(type[] arr);
```

Per la funzione `ins`, che aggiunge un solo elemento al multi-insieme, il tipo del parametro passato è `Object` oppure `Number` che è un wrapper per tutti i tipi primitivi del java. Per il metodo `insAll`, invece il tipo degli elementi contenuti nell'array da aggiungere al multi-insieme va specificato e può essere, come per i costruttori, `Object, boolean, byte, short, int, long, float, double` e `char`. Riportiamo di seguito altri esempi per illustrare l'inserimento di elementi in un multi-insieme, usando le funzioni `ins` e `insAll`:

Esempio 10

```
Lvar X = new Lvar("X", 2);
Lvar Y = new Lvar("Y", 1);
char b = 'b';
int[] ar = {1,1,2,1};
String s = "ciao";

MultiSet N = MultiSet.empty;
MultiSet R = N.ins(X);
MultiSet S = N.ins(Y).ins(X);
MultiSet W = S.insAll(ar);
MultiSet Z = R.ins(s);
```

N è il multi-insieme vuoto $\{\!\!\{\ \}\!\!\}$, R ed S sono costruiti a partire dal multi-insieme vuoto N aggiungendo nel primo caso la variabile X inizializzata con l'intero 2, mentre nel caso del multi-insieme S abbiamo aggiunto al multi-insieme vuoto N due variabili, Y e X istanziate rispettivamente ai valori 1 e 2, usando due volte la funzione `ins`. Stampando questi due multi-insiemi avremmo $R = \{\!\!\{ 2 \}\!\!\}$ e $S = \{\!\!\{ 2, 1 \}\!\!\}$. Se invece stampassimo W e Z avremmo $W = \{\!\!\{ 1, 1, 2, 1, 2, 1 \}\!\!\}$ e $Z = \{\!\!\{ ciao, 2 \}\!\!\}$, infatti partendo dal multi-insieme S che già contiene gli elementi $2, 1$ aggiungendo con il metodi `insAll` tutti gli interi dell'array `ar` otteniamo proprio W. Il multi-insieme Z è un esempio di come sia possibile aggiungere ad un multi-insieme vuoto elementi di tipo diverso, infatti partendo da R che già conteneva il valore 2 della variabile logica X, abbiamo aggiunto una stringa s.

□

Esempio 11

```
Lvar X = new Lvar();
Lvar Y = new Lvar("Y", 1);
int[] ar = {1,1,2,1};

MultiSet M = new MultiSet();
MultiSet N = MultiSet.empty;
MultiSet R = M.ins(X);
MultiSet S = M.ins(Y);
MultiSet W = S.insAll(ar);
```

il multi-insieme N è analogo a quello dell'esempio precedente e il multi-insieme M è non inizializzato. Qui la variabile logica X è non inizializzata e

Y è inizializzata con il valore 1. I multi-insiemi che contengono queste `Lvar`, $R = \{\!\{ X|M \}\!\}$, $S = \{\!\{ 1|M \}\!\}$, vengono costruiti utilizzando la funzione `ins`.

Se invece vogliamo inserire tutti gli elementi di un array in una volta sola, possiamo usare la funzione `insAll`, ottenendo ad esempio: $W = \{\!\{ 1, 1, 2, 1, Y|M \}\!\}$.

□

4.3 Altri modi per costruire multi-insiemi

Abbiamo altri due modi per costruire multi-insiemi: utilizzando dei costruttori particolari che prendono in input un altro multi-insieme, oppure una lista, una variabile o un intervallo $[p, q]$, oppure usando una funzione di utilità della classe `MultiSet`, ovvero `mkMultiSet`. I costruttori della classe `MultiSet` sono i seguenti:

```
public MultiSet(MultiSet mset);
public MultiSet(String n, MultiSet mset);
public MultiSet(Lvar l);
public MultiSet(String n, Lvar l);
public MultiSet(Lst l);
public MultiSet(String n, Lst l);
public MultiSet(int p, int q);
public MultiSet(String n, int p, int q);
```

Si noti che per ogni diverso tipo di parametro ci sono i due costruttori corrispondenti: quello che assegna il nome di default al multi-insieme creato e quello che invece gli assegna quello passato come stringa tra i parametri.

Vediamo un esempio del loro utilizzo, nel caso in cui il parametro passato sia un multi-insieme:

Esempio 12

```
Lvar X = new Lvar("X");
MultiSet N = MultiSet.empty;
MultiSet L1 = N.ins(X);
MultiSet L2 = new MultiSet("L2", L1);
int[] br = {1,3,3};
MultiSet M = new MultiSet("M", L1.insAll(br));
```

In questo esempio abbiamo X variabile logica non inizializzata e N multi-insieme vuoto. Costruiamo il multi-insieme $L1$ partendo da N aggiungendovi la variabile X utilizzando la funzione `ins` descritta nel paragrafo precedente, ovvero otteniamo $L1 = \{\!\!\{ X \}\!\!\}$. Ora possiamo costruire il nuovo multi-insieme $L2$ attraverso l'operatore **new**, invocando il costruttore a due parametri, dove il primo è la stringa $L2$, nome per questo multi-insieme e il secondo è proprio il multi-insieme $L1$, otteniamo cioè $L2 = \{\!\!\{ X \}\!\!\}$, che risulta uguale a $L1$. Oppure, facendo qualche passaggio in più possiamo costruire un multi-insieme attraverso l'operatore **new**, ma stavolta, come secondo argomento non passiamo direttamente un multi-insieme precedentemente costruito, bensì il multi-insieme risultante dall'invocazione della funzione `insAll` sul multi-insieme L. Infatti `L1.insAll(br)` restituisce un multi-insieme dato da L a cui abbiamo aggiunto tutti gli elementi dell'array di interi br, e quindi possiamo passare tale multi-insieme come parametro al costruttore, ottenenedo $M = \{\!\!\{ 1, 3, 3, X \}\!\!\}$. □

La funzione `mkMultiSet` costruisce un multi-insieme contenente n variabili logiche non inizializzate:

```
public static MultiSet mkMultiset(int n);
public static MultiSet mkMultiset(String name, int n);
```

Vediamo un esempio del suo utilizzo:

Esempio 13

```
MultiSet N1 = new MultiSet(MultiSet.mkMultiset(3));
MultiSet N2 = new MultiSet("N1",MultiSet.mkMultiset(4));
```

Stampando i due multi-insiemi sopra costruiti avremmo:

```
MultiSet_1 = {_Lvar_1,_Lvar_2,_Lvar_3}
N2 = {_Lvar_4,_Lvar_5,_Lvar_6,_Lvar_7}
```

infatti il multi-insieme $N1$, con nome di default è stato costruito con la funzione `mkMultiSet` passandogli come parametro l'intero 3, quindi il risultato è un multi-insieme contenente 3 variabili logiche non inizializzate. Analogamente viene fatto per il multi-insieme $N2$, che contiene 4 variabili logiche non-inizializzate, con l'unica differenza che per $N1$ abbiamo specificato il nome nella stringa in input.

□

4.4 I vincoli su multi-insiemi

I vincoli che possiamo imporre sui multi-insiemi sono il confronto, ovvero possiamo verificare se due o più multi-insiemi sono uguali o diversi, e l'appartenenza semplice o la non appartenenza di un oggetto ad un multi-insieme dato. La forma generale di questi vincoli binari è del tipo descritto nella definizione 9 a pagina 30, ovvero

- e_1.op(e_2);

in cui **op** è uno dei seguenti metodi **(eq, neq, in, nin)** ed e_1, e_2 sono espressioni il cui tipo dipende da op. Quindi si ha ad esempio il vincolo:

```
M.eq(N);
```

in cui e_1, e_2 sono sostituiti da M ed N, istanze della classe `MultiSet` e l'operazione `op` è il metodo `eq`, che restituisce il vincolo di uguaglianza tra i due multi-insiemi in questione, ossia:

```
public Constraint eq(Object z);
```

Analogamente si hanno i vincoli di disuguaglianza, appartenenza semplice e non appartenenza, restituiti dalle funzioni della classe `MultiSet`:

```
public Constraint neq(Object z);
public Constraint in(MultiSet multiset);
public Constraint in(Lvar lvar);
public Constraint nin(MultiSet multiset);
public Constraint nin(Lvar lvar);
```

in cui l'oggetto di invocazione del vincolo risulta essere sempre di tipo `MultiSet`, mentre il parametro passato può essere di tipo `Object` per i vincoli di uguaglianza e di disuguaglianza, oppure di tipo `MultiSet` o `Lvar` per i vincoli di appartenenza e non appartenenza.

4.4.1 Vincoli di uguaglianza e disuguaglianza

Negli esempi 14, 15 confrontiamo due multi-insiemi per determinare la loro uguaglianza o disuguaglianza.

Esempio 14

```
MultiSet M = new MultiSet("M", MultiSet.empty);
MultiSet Z = new MultiSet("Z", M.ins('b').ins('a'));
MultiSet W = new MultiSet("W", M.ins('a').ins('b'));
MultiSet N = new MultiSet("N", M.ins('a'));
```

Dati i multi-insiemi M vuoto, $Z = \{\!\{ a, b \}\!\}$, $W = \{\!\{ b, a \}\!\}$ e $N = \{\!\{ a \}\!\}$, vogliamo verificare l'uguaglianza tra Z e W attraverso l'espressione `Z.eq(W)`, ossia:

```
Solver.add(Z.eq(W));
System.out.print(Solver.boolSolve());
```

Stampando il risultato di questa unificazione utilizzando la funzione `boolSolve()` della classe `SolverClass` abbiamo risposta `true`, ovvero i due multi-insiemi Z e W sono uguali. Se invece confrontiamo Z e N ci aspettiamo risposta negativa, in quanto N contiene una sola occorrenza dell'elemento $'a'$, mentre Z ne contiene 2, infatti facendo:

```
Solver.add(Z.eq(N));
System.out.print(Solver.boolSolve());
```

l'output è `false`.

Consideriamo ora due multi-insiemi illimitati, ad esempio siano $M = \{\!\{1, 1, 2|N\}\!\}$ e $S = \{\!\{1, 2|R\}\!\}$ dove N ed R sono multi-insiemi non inizializzati.

```
int[] ar = {1,1,2};
int[] br = {1,2};
MultiSet N = new MultiSet("N");
MultiSet R = new MultiSet("R");
MultiSet M = new MultiSet("M", N.insAll(ar));
MultiSet S = new MultiSet("S", R.insAll(br));
```

Unificando i multi-insiemi M ed S e stampando il risultato di tale operazione, attraverso le espressioni:

```
Solver.add(M.eq(S));
System.out.print(Solver.boolSolve());
```

otteniamo `false` per lo stesso motivo di prima: il numero di occorrenze dello stesso elemento nei due multi-insiemi è diverso, quindi non unificano.

Infine osserviamo una situazione più particolare: siano $M = \{\!\{1, X|N\}\!\}$un multi-insieme parzialmente specificato e contenente una variabile non inizia-

lizzata X, e un multi-insieme $S = \{\!\{ 2|R \}\!\}$, dove i multi-insiemi N ed R sono gli stessi del caso precedente.

```
MultiSet N = new MultiSet("N");
MultiSet R = new MultiSet("R");
Lvar X = new Lvar("X");
MultiSet M = new MultiSet("M", N.ins(X).ins(1));
MultiSet M = new MultiSet("M", R.ins(2));
```

Ora, se invochiamo il vincolo di uguaglianza su M ed S come prima, la `boolSolve()` ci restituisce `true`. Se stampiamo dopo la risoluzione del vincolo i nostri multi-insiemi e la variabile X abbiamo:

```
X = 2
R = +{1|N}+
M = +{1,2|N}+
S = +{2,1|N}+
```

L'unificazione tra i due multi-insiemi è avvenuta nel seguente modo: assegnando il valore 2 alla variabile X e il multi-insieme $\{\!\{ 1|N \}\!\}$ al resto R del multi-insieme S, M ed N risultano uguali, entrambi con resto N non inizializzato ed elementi 1 e 2 nella parte deterministica. Per rappresentare l'output di un multi-insieme è stata introdotta la notazione $+\{\}+$.

□

Siano $M = \{\!\{ b, a, \{\!\{ a \}\!\} \}\!\}$ e $N = \{\!\{ b, a, \{\!\{ a, a \}\!\} \}\!\}$ i multi-insiemi su cui vogliamo testare la condizione di disuguaglianza, ovvero `M.neq(N)`:

Esempio 15

```
MultiSet C = new MultiSet(MultiSet.empty);
MultiSet B = new MultiSet(C.ins('a'));
MultiSet E = new MultiSet(C.ins('a').ins('a'));
MultiSet M = new MultiSet("M", C.ins(B).ins('a').ins('b'));
MultiSet N = new MultiSet("N", C.ins(E).ins('a').ins('b'));

N.output();
M.output();
Solver.add(M.neq(N));
System.out.print("\ni Multinsiemi sono diversi? "+
                 Solver.boolSolve());
```

Questa verifica restituisce `true`, infatti i due multi-insiemi hanno un numero di occorrenze diverso dell'elemento $'a'$, quindi sono diversi. L'ouput prodotto da queste poche righe di codice sarà:

```
N = +{b,a,+{a,a}+}+
M = +{b,a,+{a}+}+
i Multinsiemi sono diversi? true
```

Se invece considerassimo i multi-insiemi $M = \{\!\{ X, Y \}\!\}$ e $N = \{\!\{ a, Z \}\!\}$ dove X, Y, Z sono variabili logiche non inizializzate, imponendo il vincolo `M.neq(N)` avremmo:

```
Lvar X = new Lvar("X");
Lvar Y = new Lvar("Y");
Lvar Z = new Lvar("Z");
```

```
MultiSet C = new MultiSet(MultiSet.empty);

MultiSet M = new MultiSet("M", C.ins(X),ins(Y));
MultiSet N = new MultiSet("N", C.ins(Z),ins('a'));

Solver.add(M.neq(N));
Solver.showStore();
Solver.boolSolve();
```

La funzione `boolSolve()` restituisce `true` e attraverso la funzione `showStore()` vediamo come viene riscritto il vincolo di disuguaglianza tra i multi-insiemi considerati:

```
Store: _Y.neq(a)  _Y.neq(_Z)
```

ovvero la variabile non inizializzata Y deve essere diversa dal carattere $'a'$ e X deve essere diversa da Z, affinchè i due multi-insiemi iniziali siano diversi. Non avendo ulteriori informazioni su queste variabili, la verifica di disuguaglianza risulta positiva.

Consideriamo infine due multi-insiemi illimitati, che hanno resto diverso, ad esempio $M = \{\!\!\{ X|C \}\!\!\}$ con X variabile logica non inizializzata e $N = \{\!\!\{ 1|D \}\!\!\}$ dove C e D sono due multi-insiemi non inizializzati:

```
Lvar X = new Lvar("X");
MultiSet C = new MultiSet("C");
MultiSet D = new MultiSet("D");
MultiSet M = new MultiSet("M", C.ins(X));
```

```
MultiSet N = new MultiSet("N", D.ins(1));

Solver.add(N.neq(M));
Solver.showStore();
System.out.print(Solver.boolSolve());
```

Imponendo su questi due multi-insiemi il vincolo di disuguaglianza `N.neq(M)` la `boolSolve()` restituisce `true`, e se guardiamo cosa è stato scritto nello store per riscrivere questo vincolo, con la funzione `showStore()`, abbiamo:

```
Store: 1.neq(_X)  1.nin(JSetL.MultiSet@5224ee)
```

dove `JSetL.MultiSet@5224ee` è il multi-insieme D, infatti il vincolo di disuguaglianza, dato che C e D non sono lo stesso multi-insieme, è stato riscritto secondo la regola 6.2 (i) della tabella 2.4 a pagina 21. Quindi considerando $1 \neq X$ e $1 \notin D$ i due multi-insiemi iniziali M ed N risultano giustamente diversi.

□

4.4.2 Vincoli di appartenenza e non appartenenza

Passiamo ora a qualche esempio per i vincoli di appartenenza semplice e non appartenenza.

Consideriamo il multi-insieme $M = \{\!\{ b, a, Y \}\!\}$ dove Y è una variabile logica non inizializzata, così come X. Con l'espressione `X.in(M)` verifichiamo l'appartenenza semplice al multi-insieme M.

Esempio 16

```
int[] ar = {1,1,2};
Lvar X = new Lvar("X");
Lvar Y = new Lvar("Y");
MultiSet M = new MultiSet(MultiSet.empty.ins(Y).insAll(ar));

Solver.add(X.in(M));
Solver.solve();
X.output();
```

Stampando il valore della variabile X dopo la risoluzione del vincolo, attraverso la funzione `output()`, abbiamo il seguente risultato:

```
X = 1
```

Dato che X può assumere anche altri valori, il programma fornisce poi in modo non-deterministico anche tutte le altre soluzioni, cioè:

```
X = 1
X = 2
X = _Y
```

Proviamo ora a considerare due casi particolari: sia X una `Lvar` non inizializzata, e siano i vincoli $X \in \{\!\!\{\ \}\!\!\}$ ovvero nel multi-insieme vuoto e $X \in N$ dove N è un multi-insieme non inizializzato:

```
Lvar X = new Lvar("X");
```

```
MultiSet M = MultiSet.empty;
MultiSet N = new MultiSet("N");
```

Se imponiamo il primo dei due vincoli:

```
Solver.add(X.in(M));
Solver.solve();
X.output();
```

abbiamo un fallimento in quanto nel multi-insieme vuoto non ci sono variabili e quindi l'output del pragramma è

```
JSetL.Failure
```

Se invece imponiamo il secondo vincolo, ovvero:

```
Solver.add(X.in(N));
Solver.solve();
N.output();
```

la `solve()` non fallirà, anzi, stampando N avremo il risultato di questo vincolo, cioè:

```
N = +{_X|MultiSet_2}+
```

che è legittimo, in quanto non si conosce nè il valore della variabile, nè quello del multi-insieme dato che non è inizializzato, ma non essendo vuoto, può contenere qualcosa, in questo caso conterrà X e un resto R non inizializzato, dato da `MultiSet_2`.

□

Siano $M = \{\!\{1, 2, 3\}\!\}$ e $N = \{\!\{2, 4, 4\}\!\}$ i multi-insiemi costruiti a partire dal `MultiSet.empty` aggiungendo tutti gli elementi degli array di interi *arr* e *brr* rispettivamente e sia X una variabile logica non inizializzata:

Esempio 17

```
Lvar X = new Lvar("X");
int[] arr = {1,2,3};
int[] brr = {2,4,4};
MultiSet M = new MultiSet("M", MultiSet.empty.insAll(arr));
MultiSet N = new MultiSet ("N", MultiSet.empty.insAll(brr));
```

se volessimo verificare la duplice condizione `X.nin(M)` e `X.in(N)`, cioè X deve appartenere al multi-insieme M ma non deve appartenere ad N, dovremmo imporre i vincoli nel seguente modo:

```
Solver.add(X.nin(M));
Solver.add(X.in(N));
Solver.solve();
X.output();
```

e avremmo le seguenti soluzioni, ottenute una dopo l'altra in modo non-deterministico, come nell'esempio 16:

```
X = 4
X = 4
```

□

4.5 Operazioni di utilità sui multi-insiemi

Nella classe `MultiSet` ci sono altre funzionalità che permettono all'utente di avere informazioni su un multi-insieme, ovvero è possibile sapere il nome di un multi-insieme, quanti e quali elementi vi sono contenuti o se si tratta di un multi-insieme vuoto. Inoltre esistono funzioni che ci permettono di stampare un multi-insieme, di verificare se è completamente noto e di creare un nuovo multi-insieme concatenandone altri due, oppure partendo da `n` variabili logiche non inizializzate. Riportiamo le dichiarazioni di queste funzioni ed un esempio del loro utilizzo:

```
public String getName();
public int size();
public Vector toVector();
public void print();
public void output();
public boolean isEmpty();
public boolean known();
public boolean isGround();
public MultiSet concat(MultiSet m);
public int ariety(Object x);
```

```
public int ariety(Number z);
```

Consideriamo il multi-insieme vuoto $N = \{\!\!\{\ \}\!\!\}$ e la variabile logica non inizializzata X a partire dai quali costruiamo un nuovo multi-insieme $Z = \{\!\!\{c, c\}\!\!\}$; poi creiamo il multi-insieme $S1 = \{\!\!\{X|S\}\!\!\}$ dove S è un multi-insieme non inizializzato.

```
MultiSet N = new MultiSet("N", MultiSet.empty);
Lvar X = new Lvar("X");
MultiSet Z = new MultiSet("Z", N.ins('c').ins('c'));

MultiSet S = new MultiSet("S");
MultiSet S1 = new MultiSet("S1", S.ins(X));
```

Utilizzando la funzione di sistema `print` stampiamo alcune informazioni sui multi-insiemi appena creati: Z e $S1$, richiamando le funzioni di utilità della classe `MultiSet` che restituiscono il nome, il numero e il vettore degli elementi del multi-insieme su cui sono invocate, ovvero i metodi `getName()`, `size()` e `toVector()` rispettivamente.

```
System.out.print( "il multi-insieme " + S1.getName() +" ha "
                  + S1.size()+ " elementi");
System.out.print( "\nil multi-insieme " + Z.getName() +" ha
                  questi elementi: "+ Z.toVector());
```

L'output prodotto è il seguente:

```
il multi-insieme S1 ha 1 elementi
il multi-insieme Z ha questi elementi: [c, c]
```

Le stesse informazioni si possono ottenere in modo ancor più semplice utilizzando le funzioni `print()` e `output()`; l'unica differenza tra questi due metodi consiste nel fatto che la funzione `print()` stampa soltanto il multi-insieme, mentre la funzione `output()` stampa anche il nome del multi-insieme su cui è invocata:

```
S1.print();
Z.output();
```

L'output prodotto è:

```
+{_X|S}+          \\S1.print();
Z = +{c,c}+       \\Z.output();
```

Per verificare se un multi-insieme è vuoto oppure no possiamo usare la funzione `isEmpty()` nel seguente modo: consideriamo il multi-insieme $N = \{\!\!\{\ \}\!\!\}$ che risulta essere vuoto, e il multi-insieme $W = \{\!\!\{1, 1, 2, 1\}\!\!\}$ che invece non lo è. Invocando su di essi la funzione `isEmpty()` avremmo, nel primo caso un risultato `true`, mentre nel secondo avremmo `false`.

```
int[] ar = {1,1,2,1};
MultiSet N = MultiSet.empty;
MultiSet W = N.insAll(ar);
```

```
System.out.print("il multi-insieme " + N.getName()
                    +" è vuoto?: "+ N.isEmpty());

System.out.print("il multi-insieme " + W.getName()
                    +" è vuoto?: "+ W.isEmpty());
```

L'output è infatti il seguente:

```
il multi-insieme emptyMultiSet è vuoto?: true
il multi-insieme MultiSet_1 è vuoto?: false
```

Vediamo ora come si comportano le funzioni `isGround()` e `known()`. Abbiamo diverse situazioni: consideriamo il multi-insieme vuoto N e il multi-insieme $M = \{\!\!\{1, 1, 2, 1\}\!\!\}$, completamente noto. In entrambi i casi, applicando ad N ed M sia la funzione `isGround()` che `known()` otteniamo sempre `true` come risultato, infatti il seguente frammento di codice:

```
MultiSet N = MultiSet.empty;
int[] ar = {1,1,2,1};
MultiSet M = new MultiSet("M", ar);

System.out.print("\nil multi-insieme " + N.getName()
          +" è completamente noto?:" +N.isGround());

System.out.print("\nil multi-insieme " + N.getName()
             +" è completamente noto?:" +M.known());
```

produce in output:

```
il multi-insieme emptyMultiSet è completamente noto?: true
il multi-insieme M è completamente noto?: true
```

Se invece consideriamo il multi-insieme $R = \{\!\{ X \}\!\}$ dove X è una `Lvar` non inizializzata, invocando su R le funzioni `isGround()` e `known()` abbiamo:

```
Lvar X = new Lvar("X");
MultiSet N = MultiSet.empty;
MultiSet R = N.ins(X);

System.out.print("\nil multi-insieme " + R.getName()
         +" è completamente noto?: " +R.isGround());

System.out.print("\nil multi-insieme " + R.getName()
            +" è completamente noto?: " +R.known());
```

che restituisce il seguente risultato:

```
il multi-insieme MultiSet_2 è completamente noto?: false
il multi-insieme MultiSet_2 è completamente noto?: true
```

Infatti il multi-insieme R risulta noto in quanto contiene la variabile X, quindi il metodo `known()` restituisce `true`, ma non essendo X inizializzata, la funzione `isGround()` produce come risultato `false`. Consideriamo ora un'altra situazione in cui abbiamo a che fare con multi-insiemi illimitati. Sia ad esempio il multi-insieme $S = \{\!\{ 1, 1, 2 | Q \}\!\}$ dove Q è un multi-insieme non inizializzato.

```
int[] ar = {1,1,2};
MultiSet Q = new MultiSet();
MultiSet S = new MultiSet("S", Q.insAll(ar));
```

Se invochiamo su R le funzioni `isGround()` e `known()` all'interno della funzione di sistema `print()`, come nei casi precedenti, otteniamo il seguente output:

```
il multi-insieme S è completamente noto?: false
il multi-insieme S è completamente noto?: true
```

Infatti `R.isGround()` restituisce `false` perchè il multi-insieme in questione è illimtato, ovvero ha un resto Q non inizializzato, mentre invocando `R.known()` otteniamo come risultato `true`, perchè il multi-insieme S è parzialmente specificato. Abbiamo un'ultima situazione, ovvero se consideriamo il multi-insieme $W = \{\!\![X|Q]\!\!\}$, dove X è la variabile logica non inizializzata e Q è il multi-insieme del caso precedente, cioè un multi-insieme non inizializzato:

```
Lvar X = new Lvar("X");
MultiSet Q = new MultiSet();
MultiSet W = new MultiSet("W", Q.ins(X));
```

Invocando su W le funzioni `isGround()` e `known()` allo stesso modo dei casi precedenti, abbiamo in output:

```
il multi-insieme W è completamente noto?: false
```

```
il multi-insieme W è completamente noto?: true
```

per lo stesso motivo del caso precedente: la funzione `isGround()` restituisce `false` perchè il multi-insieme W non è completamente specificato, mentre la funzione `known()` restituisce `true` perchè W contiene almeno una `Lvar`, cioè X. Analizziamo ora il funzionamento del metodo `concat` della classe `MultiSet`: questa funzione restituisce un multi-insieme dato dalla concatenazione degli elementi del multi-insieme di invocazione e di quelli del multi-insieme passato come parametro. Questa funzione considera soltanto la parte deterministica dei due multi-insiemi coinvolti, non prende in considerazione eventuali resti, nè inizializzati, nè non inizializzati.

```
Lvar X = new Lvar("X");
int[] ar = {1,1,2};
MultiSet Q = new MultiSet();
MultiSet S = new MultiSet("S", Q.ins(X));
MultiSet W = new MultiSet("W", Q.insAll(ar));

MultiSet N = new MultiSet("N",W.concat(S));
N.output();
```

Siano X e Q una `Lvar` e un `MultiSet` non inizializzati rispettivamente. Costruiamo due nuovi multi-insiemi così fatti: $S = \{\!\!\{ X|Q \}\!\!\}$ e $W = \{\!\!\{ 1,1,2|Q \}\!\!\}$, partendo da questi creiamo un altro multi-insieme utilizzando la funzione `concat`, invocata su W e che prende in input S. Stampando il nuovo multi-insieme attraverso la funzione `output()` otteniamo:

```
N = +{1,1,2,_X}+
```

abbiamo concatenato le parti deterministiche dei due multi-insiemi in oggetto, tralasciando i resti, come ci aspettavamo. L'ultima funzione di utilità della classe `MultiSet` è `ariety` di cui abbiamo due definizioni, sostanzialmente analoghe, ma che differiscono per il parametro passato: una prende in input un `Number` e l'altra un `Object`, che hanno lo stesso significato spiegato per la funzione `ins` nel paragrafo 4.2. Questo metodo calcola il numero di occorrenze di un dato elemento in un multi-insieme, in modo deterministico. Sia $S = \{\!\{ 1, 1, 2, 3, 2, 3, 1, 1, 3 \}\!\}$ il multi-insieme in cui cerchiamo l'arietà di 1, 2 e 4.

```
int[] ar = {1,1,2,3,2,3,1,1,3};
MultiSet W = new MultiSet("W", ar);

int ar_1 = W.ariety(1);
int ar_2 = W.ariety(2);
int ar_4 = W.ariety(4);
```

Stampando i tre valori ottenuti con la funzione `ariety` abbiamo che l'arietà di 1 è 4, l'arietà di 2 è 2, mentre l'arietà di 4, che non è presente nel multi-insieme W è 0.

□

Capitolo 5

Multi-insiemi in JSetL: aspetti implementativi

JSetL è una libreria realizzata come un package Java. In questo capitolo ci occuperemo dell'implementazione della nuova classe `MultiSet`, (che va ad aggiungersi alle classi più importanti già esistenti della libreria: `Lvar`, `Set` e `Lst`) e delle modifiche alle altre classi, necessarie per l'introduzione dei multi-insiemi.

5.1 La classe MultiSet

In JSetL un multi-insieme è un oggetto della classe `MultiSet`. Questa classe è definita `public`, mentre i suoi attributi sono dichiarati `protected`. In questo modo chiunque può dichiarare riferimenti ad oggetti di tipo `MultiSet` o accedere a membri `public` della classe, ma gli attributi di questi oggetti sono accessibili solo dai metodi delle classi appartenenti al package `JSetL`.

I metodi della classe `MultiSet` sono in parte `public` cioè accessibili da tutte le classi, e in parte `protected`, ovvero accessibili solo dalle classi del

package. Ogni oggetto `MultiSet` ha diversi attributi nascosti che contengono i dati associati a quel multi-insieme e ne definiscono lo stato durante tutta l'esecuzione di un programma.

Di seguito riportiamo gli attributi di questa nuova classe:

```
protected boolean iniz = false;
protected Vector lista;
protected MultiSet resto;
protected MultiSet equ = null;
protected int id;
protected String name = null;
private static String prefix;
private static int counter;
protected static Vector nonInizMultiSet = new Vector();
protected static final MultiSet empty =
               new MultiSet("emptyMultiSet",null,null);
```

L'attributo `iniz` indica se il multi-insieme è inizializzato oppure no. Sia X un'istanza della classe `MultiSet`, se X non è inizializzata, allora $X.iniz$ è uguale a `false`, mentre se è inizializzata è uguale a `true`.

Il campo `equ` è un riferimento ad un altro oggetto `MultiSet`. Questo attributo è stato introdotto per rendere più efficiente la propagazione dei vincoli. Infatti, si supponga di aver definito n multi-insiemi istanze della classe `MultiSet`: $M_1, \ldots, M_n$. Supponiamo che essi siano tutti non inizializzati e che siano vincolati ad essere tutti uguali. Non è conveniente considerare questi n multi-insiemi come oggetti tutti distinti, è molto più naturale considerarli come riferimenti ad un unico multi-insieme, in quanto qualunque

operazione eseguita su uno di questi multi-insiemi deve avere lo stesso effetto anche su tutti gli altri; questo è reso possibile dall'attributo `equ`.

Ad esempio, se abbiamo definito i vincoli $M_i.eq(N)$ per tutti gli interi tra 1 ed n, dove N è un multi-insieme inizializzato così fatto: $\{\!|\, 1, 1, 2, 1, 3 \,|\!\}$, questo vincolo viene propagato a tutti gli n multi-insiemi, in quanto essi sono legati tra loro dalla relazione di uguaglianza. L'attributo `equ` ci permette di propagare il vincolo senza dover fare n assegnamenti passando N come parametro, cioè, se questo vincolo di uguaglianza è soddisfacibile, allora viene modificato il campo `equ` di ogni M_i assegnandogli N. Il campo `equ` è `null` di default e una volta che gli è stato assegnato un valore diverso da `null`, non può essere modificato, se non durante il backtracking. Quando il campo `equ` di un multi-insieme è un riferimento ad un altro multi-insieme significa che i due multi-insiemi sono vincolati ad essere uguali.

Tutte le volte che viene invocato un metodo su un `MultiSet` (o un `MultiSet` viene passato come parametro ad un metodo), questo metodo verifica se il campo `equ` di quel multi-insieme è uguale a `null`: se lo è, procede nell'escuzione del metodo, in caso contrario applica il metodo alla variabile a cui si riferisce il campo `equ`.

I campi `lista` e `resto` servono per memorizzare il valore del multi-insieme. L'attributo `lista` è un riferimento ad un vettore, mentre l'attributo `resto` è un riferimento ad un oggetto `MultiSet`.

L'attributo `id` permette di confrontare due multi-insiemi non inizializzati. Siano X, Y due multi-insiemi senza valore, scrivendo $X.eq(Y)$, vengono confrontati `X.id` e `Y.id` che risulteranno essere diversi, mentre paragonando X e X, ovvero $X.eq(X)$, vengono confrontati `X.id` e `X.id` che sono uguali.

I campi `name`, `prefix` e `counter` servono per poter assegnare un nome ad ogni multi-insieme che viene creato: se il nome viene esplicitamente specificato nella dichiarazione, con esso verrà settato l'attributo `name`, altrimenti viene creato un nome apposito da assegnare al multi-insieme; esso è dato dalla stringa `MultiSet_`, valore di default di `prefix` e un numero progressivo memorizzato in `counter`.

La classe `MultiSet` possiede, poi, l'attributo `empty`. Questo attributo è dichiarato `static`, cioè ne esiste un'unica copia indipendentemente dal numero di oggetti `MultiSet` che vengono creati, inoltre `empty` è dichiarato `final`, cioè non può essere modificato. Questo campo definisce il multi-insieme vuoto; esso è un riferimento ad un multi-insieme i cui campi `lista` e `resto` sono uguali a `null`, come accade per un multi-insieme non inizializzato, ma il cui campo `iniz` è settato a `true` dalla funzione protetta che viene richiamata tutte le volte che viene costruito il multi-insieme vuoto:

```
protected MultiSet(String n, Vector v, MultiSet mset);
```

Infine la classe `MultiSet` possiede un altro attributo statico, dato dal vettore `nonInizMultiSet`, i cui elementi sono riferimenti a multi-insiemi non inizializzati oppure che erano tali al momento della loro creazione. Questo vettore viene utilizzato quando viene creato, durante la risoluzione dei vincoli, un punto di scelta.

Ad esempio, consideriamo i multi-insiemi M, N, P, Q e Z definiti come segue:

Esempio 18

```
int[] ar = {1,1,2};                          //array di interi
MultiSet M = new MultiSet();               //M non inizializzato
MultiSet N = M.ins(2);                            //N = +{2|M}+
MultiSet P = M.insAll(ar);                     //P = +{1,1,2|M}+
MultiSet Q = MultiSet.empty;                           //Q = +{}+
MultiSet Z = new MultiSet("Z", ar);          //Z = +{1,1,2}+
```

`M.lista` e `M.resto` sono entrambi `null` in quanto `M` è un multi-insieme non inizializzato. Il campo `N.lista` è un riferimento ad un vettore che contiene solo l'intero 2, `N.resto` è un riferimento al multi-insieme `M`. L'attributo `P.lista` è un riferimento al vettore contenente due occorrenze di 1 e una di 2, mentre `P.resto` è un riferimento al multi-insieme `M`. I campi `Q.lista` e `Q.resto` sono entrambi `null` in quanto `Q` è il `MultiSet` vuoto. Infine il campo `Z.lista` è dato dal vettore `ar` e il suo `resto` è il `MultiSet` vuoto.

□

Quindi un multi-insieme `N` è limitato se `N.resto` è il multi-insieme vuoto oppure è un multi-insieme limitato a sua volta; mentre è non limitato se `N.resto` è un multi-insieme non inizializzato o non limitato.

5.2 Inserimento ed estrazione da multi-insiemi

Nella classe `MultiSet`, come nelle altre classi del package JSetL, `Lst` e `Set`, sono previsti diversi metodi `public` che permettono di aggiungere o rimuovere

uno o più elementi dal multi-insieme. Nessuno di questi metodi modifica il multi-insieme su cui è invocato, tutti costruiscono e restituiscono un nuovo multi-insieme.

5.2.1 Inserimento di elementi in un multi-insieme

I metodi che consentono l'inserimento di uno o più elementi in un multi-insieme sono i seguenti:

```
public MultiSet ins(type elem);
public MultiSet insAll(type[] arr);
```

In Java è possibile effettuare l'overloading dei metodi, quindi abbiamo potuto sovraccaricare questi metodi con diverse implementazioni che si differenziano soltanto per il tipo del parametro passato. Infatti con `type` vengono indicati i seguenti tipi: `Number`, (che è un wrapper per i tipi primitivi quali: `boolean, char, int, float, short, long, double, byte`) e `Object` che invece è una superclasse per i tipi generici, tra cui `Lvar, Lst, Set` e `MultiSet`.

Riportiamo di seguito l'implementazione del metodo `ins`, della classe `MultiSet`, nel caso in cui il parametro passato sia un oggetto generico; negli altri casi l'implementazione è del tutto simile:

```
public MultiSet ins(Object o) {
    if(this.equ == null) {
        Vector v = new Vector();
        v.add(o);
        MultiSet m = new MultiSet(v, this);
```

```
            return m;
        }
        else return this.equ.ins(i);
    }
```

Questo metodo definisce un vettore vuoto `v` al quale viene aggiunto il parametro passato. Se il parametro è di tipo primitivo, viene prima trasformato in un oggetto della corrispondente classe avvolgente. Il metodo crea e restituisce un nuovo `MultiSet m` tale che `m.lista` è un riferimento a `m` e `m.resto` è un riferimento a `this`, cioè all'insieme su cui è stato invocato il metodo.

Il metodo `insAll` è del tutto analogo, ma anzichè aggiungere al vettore `v` un solo elemento, aggiunge tutti gli elementi dell'array passato come parametro, usando un ciclo `for`:

```
    public MultiSet insAll(int[] arr) {
        if(this.equ == null) {
            Vector v = new Vector();
            for(int i = 0; i < arr.length; i++){
                Integer o = new Integer(arr[i]);
                v.add(o);
            }
            MultiSet l = new MultiSet(v, this);
            return l;
        }
        else return this.equ.insAll(arr); }
```

5.2.2 Estrazione di elementi da un multi-insieme

I metodi per l'estrazione di uno o più elementi da un multi-insieme sono i seguenti:

```
protected MultiSet sub(Object z);
protected MultiSet subAll(Object z);
protected MultiSet subfirst();
protected MultiSet sub();
```

Il metodo `sub(Object z)` serve per eliminare la prima occorrenza di `z` dal multi-insieme su cui è invocato, mentre `subAll(Object z)` rimuove tutte le occorrenze di `z`. Le funzioni `subfirst()` e `sub()` restituiscono il multi-insieme su cui è stata invocata, eliminando il primo elemento; l'unica differenza riguarda il multi-iniseme di invocazione, ovvero mentre la `subfirst()` lavora soltanto sulla parte specificata del multi-insieme, la `sub()` considera l'intero multi-insieme, sia la parte inizializzata che l'eventuale resto. Questi metodi sono tutti `protected`, ovvero sono accessibili solo dall'interno del package JSetL, perchè sono deterministici, ovvero non tengono conto di tutti i possibili casi che derivano dall'eliminazione di un oggetto all'interno di un multi-insieme, ad esempio non lavorano bene in presenza di variabili non inizializzate. Riportiamo di seguito l'implemetazione del metodo `sub(Object z)`, per meglio comprendere come funzionano questi quattro metodi, molto simili tra loro:

```
protected MultiSet sub(Object z) {
   if(this.equ == null) {
     if(this.lista.contains(z)) {
```

```
            Vector v = new Vector();
            v.addAll(this.lista);
            v.remove(z);
            if(v.size() == 0) return this.resto;
            else {
               MultiSet s = new MultiSet(v, this.resto);
               return s;
            }
         }
         else{
           MultiSet ss = new MultiSet(this.lista, this.resto.sub(z));
           return ss;
         }
       }
      else return this.equ.sub(z);
   }
```

Questo metodo, come tutti gli altri metodi utili per inserimento e cancellazione di elementi all'interno di un multi-insieme, controlla che il `MultiSet`, oggetto di invocazione non sia un riferimento ad un altro multi-insieme, verificando che il campo `equ` sia `null`. In caso contrario, la procedura risolve prima la catena di riferimenti e poi applica effettivamente la rimozione dell'elemento.

Questi metodi sfruttano alcune procedure proprie della classe `Vector` di Java: infatti dato che `this.lista` è un `Vector`, è possibile invocare sul vettore `v`, che viene creato in questo metodo, la funzione `addAll` che prende come parametro proprio `this.lista`. Inoltre su `v` vengono invocate le funzioni `remove` e `size`, che rimuovono un elemento all'interno del vettore e restituiscono il numero di elementi contenuti in esso, rispettivamente.

Ognuno di questi metodi per l'eliminazione degli elementi all'interno di un multi-insieme, restituisce un nuovo `MultiSet`, ottenuto da quello di partenza con la cancellazione dell'elemento passato come parametro.

I metodi `subfirst` e `sub` servono per eliminare l'elemento che nel multi-insieme si trova in prima posizione.

5.3 Vincoli su multi-insiemi

I vincoli che sono stati implementati sono: uguaglianza, disuguaglianza, appartenenza semplice e non appartenenza. Utilizzando le regole di riscrittura descritte in dettaglio nel Capitolo 2, sono stati implementati i relativi algoritmi per consentire la risoluzione dei vincoli anche in presenza di multi-insiemi.

Secondo il meccanismo utilizzato in JSetL per il trattamento dei vincoli, descritto nel paragrafo 3.5.3 a pagina 43, i metodi di riscrittura dei vincoli vengono richiamati, in base al tipo degli argomenti coinvolti in un dato vincolo, da:

```
protected void eq(Constraint s)throws Failure;
protected void neq(Constraint s)throws Failure;
protected void in(Constraint s)throws Failure;
protected void nin(Constraint s)throws Failure;
```

metodi di `RewritingConstraintsRules`, che risolvono i vincoli di uguaglianza, disuguaglianza, appartenenza semplice e non appartenenza rispettivamente.

Nella classe `RewritingConstraintsRules` sono state aggiunte le definizioni dei seguenti metodi:

```
protected void eqMultiSet(MultiSet mset1, MultiSet mset2,
                          Constraint s) throws Failure;
protected void eqLvarMultiSet(Lvar l, MultiSet mset,
                          Constraint s) throws Failure;
protected void eqMultiSetObj(MultiSet mset, Object o,
                          Constraint s) throws Failure;
protected void neqMultiSet(MultiSet mset1, MultiSet mset2,
                          Constraint s) throws Failure;
protected void neqLvarMultiSet(Lvar l, MultiSet mset,
                          Constraint s) throws Failure;
protected void neqMultiSetObj(MultiSet mset, Object o,
                          Constraint s) throws Failure;
protected void inObjMultiSet(Object obj, MultiSet multiset,
                          Constraint s) throws Failure;
protected void inLvarMultiSet(Lvar lvar, MultiSet multiset,
                          Constraint s) throws Failure;
protected void ninLvarMultiSet(Lvar lvar, MultiSet mset,
                           Constraint s)throws Failure;
protected void ninObjMultiSet(Object obj, MultiSet mset,
                          Constraint s) throws Failure;
```

I vincoli di uguaglianza e di disuguaglianza sono implementati attraverso tre funzioni diverse, che si differenziano per gli argomenti da confrontare: `eqLvarMultiSet` e `neqLvarMultiSet` paragonano una variabile logica ed un multi-insieme, ovvero se il vincolo da trattare è del tipo $X.eq(M) \backslash X.neq(M)$ con X variabile logica e M multi-insieme.
I metodi `eqMultiSetObj` e `neqMultiSetObj` confrontano un multi-insieme

ed un oggetto generico che non sia nè un multi-insieme, nè una variabile logica. Infine se gli oggetti da confrontare sono entrambi dei multi-insiemi, ad esempio in presenza del vincolo $M.eq(N) \backslash M.neq(N)$ con M ed N istanze della classe `MultiSet` allora le due funzioni coinvolte sono `eqMultiSet` e `neqMultiSet` rispettivamente. In realtà il metodo `eqMultiSetObj` restituisce sempre un fallimento, in quanto è impossibile confrontare un oggetto di tipo `Lst`, `Set` oppure un oggetto generico con un `MultiSet`, mentre il metodo `neqMultiSetObj` restituisce sempre `true`.

Per i vincoli di appartenenza semplice e di non appartenenza, ovvero in presenza dei vincoli `in` e `nin`, si distingue il caso $X.in(M)$, con X una istanza della classe `Lvar` e M istanza della classe `MultiSet`, in cui viene invocato il metodo `inLvarMultiSet`, da $Y.in(M)$, con Y oggetto della classe `Object`, cioè diverso da una variabile logica, da un insieme, da una lista e da un multi-insieme, con M sempre istanza della classe `MultiSet`, in cui invece, viene invocata la funzione `inObjMultiSet`. Discorso analogo avviene nei casi $X.nin(M)$, e $Y.nin(M)$.

Per poter completare il discorso di trattamento di vincoli in presenza di multi-insiemi, sono stati modificati i metodi `inLvarLvar` e `inObjLvar` della classe `RewritingConstraintsRules`, aggiungendo un controllo sul tipo del secondo argomento: ovvero se il valore del secondo argomento fosse un'istanza della classe `MultiSet`, allora viene invocato il metodo `inLvarMultiSet` nell'implementazione di `inLvarLvar`, mentre si richiama `inObjMultiSet` in quella di `inObjLvar`.

I vincoli di uguaglianza e di disuguaglianza utilizzano due particolari funzioni ausiliarie: **tail** e **untail**, definite al paragrafo 2.4.1 a pagina 17. Queste

importanti funzioni sono state definite nella classe `MultiSet`:

```
protected MultiSet ultimoResto();      //funzione tail
protected MultiSet notultimoResto();   //funzione untail
```

Ecco come procede la funzione `ultimoResto()`: prima controlla che il MultiSet, oggetto d'invocazione, non sia un riferimento ad un altro MultiSet; perchè in tal caso richiama sè stessa sul multi-insieme a cui fa riferimento. Se il campo `this.equ` è `null`, invece, questa funzione restituisce l'oggetto di invocazione, se esso non era inizializzato (primo ramo `if`), oppure restituisce `this.resto` se il multi-insieme di partenza aveva il `resto` vuoto, uguale a `null` oppure era non inizializzato (secondo ramo `if`); altrimenti, (ramo `else`) richiama la funzione sul campo `this.resto`.

```
protected MultiSet ultimoResto() {
  if (this.equ == null) {
    if (!this.iniz) return this;
    else if (this.resto == empty || this.resto == null ||
             !this.resto.iniz || this.resto.isEmptySL())
                    return this.resto;
    else return this.resto.ultimoResto();
    }
  else return this.equ.ultimoResto();
}
```

Sia il multi-insieme $M = \{\!\{1, 2, 1|R\}\!\}$, applicandovi la funzione `ultimoResto()`, viene superato il primo controllo e si entra nel secondo ramo dell'`if`: se il

multi-insieme R fosse vuoto, ovvero $R = \{\!\!\{\ \}\!\!\}$, oppure fosse composto da `Lvar` non inizializzate, ad esempio in presenza del seguente frammento di codice:

```
Lvar X = new Lvar();          //X e Y variabili logiche non
Lvar Y = new Lvar();          //inizializzate

MultiSet R = new MultiSet(MultiSet.empty.ins(X).ins(Y));
                              // R ={[X,Y]}
```

allora la funzione restituisce il multi-insieme R. Se invece il campo `resto` fosse `null`, cioè il multi-insieme M fosse completamente specificato, ad esempio $M = \{\!\!\{ 1, 2, 1 \}\!\!\}$; allora la funzione `ultimoResto` (il secondo ramo `if`) darebbe come risultato `null`.

Il ramo `else` invece traduce la seguente situazione: siano il multi-insieme $M = \{\!\!\{ 1, 2, 1 | R \}\!\!\}$, il multi-insieme $R = \{\!\!\{ 2 | S \}\!\!\}$, ovvero R è inizializzato, dove S è un altro multi-insieme. La funzione `ultimoResto()` applicata ad M, vede che `M.resto` è uguale al multi-insieme R, quindi richiama sè stessa su R, che a sua volta ha `resto` uguale a S, che sarà il `MultiSet` risultante, cioè l'*ultimo resto* del multi-insieme M.

La funzione `notultimoResto()` invece procede in modo diametralmente opposto; ovvero considerando i multi-insiemi $M = \{\!\!\{ 1, 2, 1 | R \}\!\!\}$, $R = \{\!\!\{ X, Y \}\!\!\}$, con X e Y variabili logiche non inizializzate essa restituisce un nuovo `MultiSet` così fatto: $\{\!\!\{ 1, 2, 1 \}\!\!\}$; nel secondo caso, ovvero in presenza di $M = \{\!\!\{ 1, 2, 1 | R \}\!\!\}$, $R = \{\!\!\{ 2 | S \}\!\!\}$, dove S è un altro multi-insieme che assumiamo non essere inizializzato, la funzione `notultimoResto()` produce il seguente risultato: $v = \{1, 2, 1, 2\}$ cioè inserisce tutti gli elementi noti di M e del suo `resto` R, in un vettore v. Questa funzione utilizza infatti un'altra importante procedu-

ra della classe `MultiSet`, la `toVector()`, che produce un vettore contenente tutti gli elementi del multi-insieme su cui viene invocata.

Nei paragrafi 5.5.1 e 5.5.2 analizzeremo in dettaglio l'implementazione del vincolo di uguaglianza nel caso di due multi-insiemi e del vincolo di appartenenza semplice in presenza di una variabile logica ed un multi-insieme.

5.4 Ulteriori modifiche a JSetL

Per permettere la coesistenza in JSetL di multi-insiemi e altre stutture dati, quali variabili logiche, liste ed insiemi e consentire la risoluzione di vincoli che coinvolgono diverse istanze di questi oggetti, sono state modificate altre classi della libreria oltre a `RewritingConstraintsRules`: sono state apportate modifiche a `VarState`, `SolverClass` e `Backtracking`.

Nella classe `VarState` abbiamo aggiunto l'attributo `multisetNotInit` di tipo `Vector` che contiene un riferimento ad un vettore di multi-insiemi non inizializzati, utile in caso di backtracking e soluzioni non-deterministiche, come già spiegato nel paragrafo 3.5.1 a pagina 38. Di conseguenza abbiamo modificato sia il costruttore di default che il costruttore con parametri, per inizializzare questo nuovo membro. L'altra modifica riguarda l'unica funzione della classe `VarState`, `cloneVarState()`, che memorizza lo stato in cui si trovano tutte le variabili, insiemi, liste e multi-insiemi non inizializzati, nel momento in cui si è creato un punto di scelta per un'altra possibile soluzione dei vincoli:

```
protected VarState cloneVarState() {
  VarState vs = new VarState();
```

```
    vs.lvarNotInit = (Vector)((Vector)this.lvarNotInit).clone();
    vs.setNotInit  = (Vector)((Vector)this.setNotInit).clone();
    vs.lstNotInit  = (Vector)((Vector)this.lstNotInit).clone();
    vs.multisetNotInit  = (Vector)((Vector)this.multisetNotInit).
                                                        clone();
    return vs; }
```

Nella classe `SolverClass` abbiamo aggiunto una nuova definizione per il metodo `setof` che, coerentemente con ciò che succede per gli insiemi, restituisce un multi-insieme composto da tutte le possibili soluzioni di un insieme di vincoli. E' possibile infatti, in presenza di multi-insiemi che una collezione di vincoli abbia più soluzioni trovate in modo non-deterministico e che alcune di esse siano ripetute, quindi è necessario avere una funzione analoga a quella che si aveva per gli insiemi, ma che restituisca un multi-insieme. Questo metodo è `setof` ed è così definito:

```
    public MultiSet setof(MultiSet ob) throws Failure {
            return multisetofSL(ob);
        }
```

Il metodo richiama un'altra funzione, `multisetofSL`, che crea un vettore `allValues` in cui memorizza tutte le soluzioni dell'insieme dei vincoli. Questo vettore verrà passato come parametro al costruttore della classe `MultiSet`, assieme al `MultiSet.empty` per creare il multi-insieme risultato.

```
 private MultiSet multisetofSL(Object ob) throws Failure {
   Vector allValues = new Vector();
```

```
  finalSolve();
  Object newO;
  if (ob instanceof Lvar) newO = ((Lvar)ob).clona();
  else if (ob instanceof Lst) newO = ((Lst)ob).clona();
  else if (ob instanceof MultiSet) newO = ((MultiSet)ob).clona();
  else newO = ((Set)ob).clona();

  allValues.add(newO);

  while (B.alternatives.size() != 0) {
  B.gestisciFallimentoSetof();
  if (solveSetof()) {
      Object newOcp;
      if (ob instanceof Lvar) newOcp = ((Lvar)ob).clona();
      else if (ob instanceof Lst) newOcp = ((Lst)ob).clona();
      else if (ob instanceof MultiSet) newOcp = ((MultiSet)ob).
                                                    clona();
      else newOcp = ((Set)ob).clona();
      allValues.add(newOcp);
       }
   }
  return new MultiSet(allValues, MultiSet.empty);
}
```

Questo metodo richiama la `finalSolve()` che assomiglia alla `solve()`. Quando viene trovata una soluzione, essa viene memorizzata nel vettore `allValues` e viene forzato un fallimento fino a quando non sono state esplorate tutte le alternative rimaste aperte dopo l'esecuzione della `finalsolve()`. A seconda che ogni soluzione sia di tipo `Lvar, Lst, Set` o `MultiSet` viene invocata la giusta funzione `clona()` per ricreare una copia perfetta dell'ogget-

to.

Infine, nella classe `Backtracking` sono state modificate tre funzioni per permettere il corretto ripristino della situazione delle variabili, dopo un fallimento, anche in presenza di multi-insiemi. E' stata modificata `getVarState()` che restituisce un elemento di tipo `StatoVar` che ha come attributi quattro vettori: un vettore di `Lvar`, uno di `Set`, uno di `Lst` e uno di `MultiSet`. In ciascuno di questi vettori vengono memorizzate le variabili che non sono ancora inizializzate. In questo modo se in seguito, durante il backtracking, sarà necessario tornare ad uno stato precedente, le variabili che prima del fallimento che ha innescato il backtracking erano non inizializzate torneranno tali. Un'altra modifica che si è resa necessaria riguarda il metodo `restoreVarState()` che ripristina i valori di default per tutte queste variabili non inizializzate.

Anche il metodo `updateVarAlternative()` della classe `Backtracking`, che viene richiamato dalla `solve()` della classe `SolverClass` è stato modificato, per poter aggiornare completamente tutti i punti di scelta dello stack delle alternative, aggiungendo tutte le variabili non inizializzate definite dopo la crezione di un'alternativa, quindi tenendo conto anche di quelle variabili che sono istanze della classe `MultiSet`.

5.5 Implementazione degli algoritmi di unificazione ed appartenenza

5.5.1 Vincolo di uguaglianza tra multi-insiemi

Analizziamo ora più in dettaglio il metodo

```
protected void eqMultiSet(MultiSet mset1, MultiSet mset2,
                          Constraint s) throws Failure;
```

che implementa l'algoritmo di unificazione tra due multi-insiemi, di cui abbiamo parlato nel paragrafo 2.4.1 a pagina 17.

Il metodo comincia con una condizione `if then else`; viene controllato infatti che i due multi-inisemi da confrontare non siano entrambi dei riferimenti ad altri multi-insiemi. Questa verifica viene fatta analizzando il campo `equ` sia del `mset1` che del `mset2`. Se uno dei due o entrambi sono diversi da `null`, allora viene richiamata questa stessa funzione sui multi-insiemi a cui quelli di partenza si riferiscono. Nel caso in cui invece `mset1.equ == null && mset2.equ == null` si apre uno `switch` in cui vengono tradotte le regole dell'algoritmo di unificazione. Vediamo nel seguito i diversi `case` dello `switch` che corrispondono alle regole della tabella 2.3.

Nel caso in cui il `caseControl` del `Constraint s` abbia valore 0, cioè all'inizio della risoluzione del vincolo di uguaglianza, si entra nel primo blocco `case`

```
case 0:                                          //  (1)
        if(mset1 == mset2) {
            s.bool = true;
            return;
        }
        else if(!mset1.iniz && !mset2.iniz) {
            mset2.iniz = true;
            mset2.equ = mset1;
```

```
    s.arg2 = mset1;
    s.bool = true;
    Solver.storeInvariato = false;
    return;
}
else if(!mset1.iniz) {                //  (2)
    if(mset2 == MultiSet.empty) {
    mset1.equ = mset2;
    Solver.storeInvariato = false;
    return;
    }
    else if(mset2.occurs(mset1)) {
    Solver.fail(s);
    return;
    }
    else {                            //  (3)
    if(mset2.ultimoResto() == mset1 && !mset2.isEmpty()){
                    MultiSet N = new MultiSet();
        MultiSet M = new MultiSet(mset2.toVector(), N);
        s.arg2 = M;
        eqMultiSet(mset1, M, s);
        Solver.storeInvariato = false;
        return;
    }
    else {
            mset1.iniz = true;
        mset1.equ = mset2;
        s.bool = true;
        Solver.storeInvariato = false;
        return;
    }
```

```
    }
}
```

La regola (1) è tradotta verificando se i due multi-insiemi in oggetto sono uguali oppure sono entrambi non inizializzati, in tal caso viene restituito `true` e la funzione termina. Poi si controlla se invece solo uno dei due risulta senza valore, ad esempio il primo `mset1`. In questo caso, ovvero nel caso (2) dell'algoritmo di unificazione sono possibili diverse situazioni che vengono trattate con una sequenza di clausole `if then else`. La prima risolve il caso in cui `mset1` è non inizializzato e `mset2` è il multi-insieme vuoto, situazione che viene riscritta assegnando il secondo multi-insieme come riferimento al primo. Se invece `mset1` è contenuto in `mset2` allora l'unificazione fallisce con un `fail` e la computazione termina. Infine se `mset2.ultimoResto() == mset1 && !mset2.isEmpty()` cioè il resto del secondo multi-insieme, che non è vuoto, è uguale al primo multi-insieme, allora stiamo traducendo il caso $X.eq(\{\!\!\{ t_0, \ldots, t_n \mid X \}\!\!\})$ ossia il caso (3) della tabella 2.3. In questa situazione rilanciamo il metodo `equMultiSet` passandogli come parametri il multi-insieme `mset1` e un nuovo multi-insieme formato da tutti gli elementi di `mset2` e un resto nuovo senza valore che chiamiamo `N`, ovvero scriviamo il nuovo vincolo nel constraint store $X.eq(\{\!\!\{ t_0, \ldots, t_n \mid N \}\!\!\})$. Analogamente ripetiamo le stesse considerazioni immaginando che `mset2` sia non inizializzato.

Un altro caso che si conclude con un fallimento è il caso (4) dell'algoritmo: se uno dei due multi-insiemi è contenuto propriamente nell'altro, è chiaro che i due multi-insiemi non unificano. Questa situazione è implementata nel seguente `else`:

```
 else {
  if(mset1.occurs(mset2) || mset2.occurs(mset1)) { //  (4)
  Solver.fail(s);
  return;
  }
```

Analizziamo ora i casi più interessanti, ovvero le regole (7) e (8) della tabella 2.3:

```
 //(7)
else if(mset1.ultimoResto() != mset2.ultimoResto() ||
          (mset1.ultimoResto().equals(MultiSet.empty)
           && mset2.ultimoResto().equals(MultiSet.empty))) {
      s.caseControl = 1;
      eqMultiSet(mset1, mset2, s);
      return;
 }
 //(8)
else if(mset1.ultimoResto() == mset2.ultimoResto()
        && mset1.ultimoResto() != MultiSet.empty) {
              s.caseControl = 3;
              eqMultiSet(mset1, mset2, s);
              return;
          }
```

Nella regola (7) avevamo la situazione $\{\!\{t \mid S\}\!\}.eq(\{\!\{t' \mid S'\}\!\})$ in cui `tail(S)` e `tail(S')` non erano la stessa variabile; questa situazione viene implementata confrontando i resti di `mset1` e `mset2` attraverso la funzione `ultimoResto()` della classe `MultiSet`; se questi risultano essere diversi, oppure sono entrambi il multi-insieme vuoto allora viene impostato il `caseControl`

a 1 e viene rilanciata la `eqMultiSet` con i multi-insiemi iniziali. Se invece i resti di `mset1` e `mset2` sono uguali ed entrambi diversi del multi-insieme vuoto, allora stiamo riscrivendo la regola (8), e come prima rilanciamo `eqMultiSet` con i multi-insiemi iniziali, ma questa volta impostiamo il campo `caseControl` a 3. Con il `caseControl` settato a 1 entriamo nel blocco `case 1` dello `switch`:

```
case 1:                                        // (7)(i)
 Solver.B.addChoicePoint(2, (VarState)Solver.B.getVarState(),
                         (StoreState)Solver.B.getStoreState(s));
 s.arg1 = mset1.first();                         // t.eq(t')
 s.arg2 = mset2.first();
 s.caseControl = 0;
 Constraint se0 = new Constraint(mset1.sub(), Enviroment.eqCode,
                                      mset2.sub());
 Solver.add(Solver.indexOf(s)+1, se0);       // S.eq(S')
 eq(s);
 Solver.storeInvariato = false;
 return;
```

Qui, come prima cosa settiamo il `caseControl` a 2 aggiungendo un punto di scelta e memorizziamo lo stato dello store e delle variabili richiamando le funzioni `getVarState()` e `getStoreState(s)` della classe `Backtracking`. Questo viene fatto per poter gestire nel blocco `case 2` dello `switch` il caso (7)(ii), mentre nel `case 1` viene trattato il caso (7)(i). In quest'ultima situazione la regola della tabella 2.3 imponeva di uguagliare t e t', e questo viene fatto lanciando `eq(s)`, dove il vincolo `s` viene costruito estraendo il primo elemento da entrambi i multi-insiemi iniziali per inizializzare `arg1` e `arg2` e riportando il valore del `caseControl` a 0. Inoltre la regola diceva di

aggiungere nel constraint store un nuovo vincolo così fatto: $S.eq(S')$. Questo si ottiene invocando il costruttore della classe `Constraint` passandogli come parametri due multi-insiemi ottenuti da quelli di partenza eliminando il primo elemento attraverso la funzione `sub()` della classe `MultiSet` e sempre `eqCode`. L'aggiunta di questo vincolo chiamato `se0` avviene attraverso il metodo `add` della classe `SolverClass`.

Il caso (7)(ii) della tabella 2.3 viene risolto nel seguente blocco dello `switch`:

```
case 2:                                    // (7)(ii)
 MultiSet n = new MultiSet();
 s.arg1 = mset1.sub();
 s.arg2 = n.ins(mset2.first());
 s.caseControl = 0;
 Constraint se1 = new Constraint(n.ins(mset1.first()),
                       Enviroment.eqCode, mset2.sub());
 Solver.add(Solver.indexOf(s)+1, se1);
 eq(s);
 Solver.storeInvariato = false;
 return;
```

Qui si doveva uguagliare il primo multi-insieme privato del primo elemento, `mset1.sub()`, e un nuovo multi-insieme formato dal primo elemento del secondo multi-insieme di partenza ed un resto non inizializzato ovvero $\{\!|\, t \mid N \,|\!\}$. Inoltre dovevamo, secondo la regola, aggiungere un nuovo vincolo `se1` del tipo $\{\!|\, t \mid N \,|\!\}.eq(s')$; lo facciamo allo stesso modo del caso (7)(i).

Nell'ultimo caso della tabella 2.3, ossia il caso (8), si verifica se `tail(S)` e `tail(S')` sono la stessa variabile, cioè $\{\!\!\{ t_1, \ldots, t_n \mid X \}\!\!\}.eq(\{\!\!\{ s_1, \ldots, s_m \mid X \}\!\!\})$. La regola dice che bisogna inserire nel constraint store il nuovo vincolo, `ss5` dato da $\{\!\!\{ t_1, \ldots, t_n \}\!\!\}.eq(\{\!\!\{ s_1, \ldots, s_m \}\!\!\})$, questo si ottiene sfruttando la funzione `notultimoResto()` sia su `mset1` che su `mset2`.

```
case 3:          // (8)
 MultiSet w = mset1.notultimoResto();
 MultiSet z = mset2.notultimoResto();

 Constraint ss5 = new Constraint(w, Enviroment.eqCode, z);

 Solver.add(Solver.indexOf(s), ss5);
 s.bool = true;
 Solver.storeInvariato = false;
 return.
```

5.5.2 Vincolo di appartenenza di una variabile logica ad un multi-insieme

Un altro vincolo importante è l'appartenenza semplice di una variabile logica ad un multi-insieme, che viene risolto nel metodo:

```
protected void inLvarMultiSet(Lvar lvar, MultiSet multiset,
                              Constraint s) throws Failure;
```

Le regole di riscrittura per questo vincolo sono descritte al paragrafo 2.4.3 a pagina 22. Vediamo ora come sono state implementate:

```
 if(multiset.iniz){
     if(multiset == MultiSet.empty) {
         Solver.fail(s);
         return;
```

Se il multi-insieme in cui si deve cercare la variabile logica in input, è vuoto, si ha un fallimento e la computazione termina. Altrimenti, anche qui, come nel metodo che implementa le regole di unificazione tra due multi-insiemi, abbiamo un costrutto `switch`, dove ora saranno presenti 2 blocchi `case` relativi alle due possibili situazioni derivanti dalla regola (2) della tabella 2.5:

```
case 0:                              // (2) (a)
 VarState statoVarIn = Solver.B.getVarState();
 StoreState statoStoreIn = Solver.B.getStoreState(s);
 if(multiset.size() > 1 || !multiset.ultimoResto().iniz)
     Solver.B.addChoicePoint(1, statoVarIn, statoStoreIn);
     s.arg1 = lvar;
     s.cons = Enviroment.eqCode;
     s.arg2 = multiset.first();
     eq(s);
     Solver.storeInvariato = false;
     return;
case 1:                              // (2) (b)
 s.caseControl = 0;
 s.arg2 = (MultiSet)multiset.sub();
 in(s);
 Solver.storeInvariato = false;
 return;
```

Stiamo implementando la regola (2) se il vincolo con cui abbiamo a che fare è del tipo: $\{\!|\, t \mid s \,|\!\}.in(r)$. Nel primo caso, al solito memorizziamo lo stato dello store e delle variabili, per il backtracking futuro, poi, se il resto del multi-insieme di partenza non ha valore oppure il multi-insieme ha elementi, aggiungiamo un punto di scelta passando il nuovo valore di `caseControl` uguale a 1 e lanciamo `eq(s)` con `s` vincolo in cui il primo argomento è dato dalla variabile logica iniziale, il secondo argomento invece è il primo elemento del multi-insieme di partenza, che è stato ricavato attraverso la funzione `first()` della classe `MultiSet`, e un `eqCode`. Abbiamo così riscritto il vincolo di appartenenza come un vincolo di uguaglianza $X.eq(t)$. Il secondo caso, invece riscritto nel secondo blocco `case`, ripristina, innanzitutto il valore del campo `caseControl` a 0, poi riscrive il vincolo di appartenenza iniziale in un altro vincolo di appartenenza così fatto: $X.in(R)$ dove X è la variabile logica iniziale ed R è il multi-insieme che si trova da quello in input eliminando il primo elemento, sfruttando la funzione `sub()` della classe `MultiSet`.

La regola (4) della tabella 2.5 è implementata in una clausola `else`, che viene invocata se il multi-insieme iniziale non è inizializzato:

```
MultiSet n = new MultiSet();
      s.arg1 = multiset;
      s.arg2 = n.ins(lvar);
      s.cons = Enviroment.eqCode;
      s.caseControl = 0;
      eq(s);
      Solver.storeInvariato = false;
      return;
```

ovvero se stiamo cercando di risolvere il vincolo $X.in(r)$ dove X è il multi-insieme di partenza. Questo vincolo viene risolto nel seguente modo$X.eq(\{\!|\, r \mid N \,|\!\})$, cioè con un vincolo di uguaglianza, dove il primo argomento resta il multi-insieme su cui è stato invocato il metodo, il secondo argomento è dato da un nuovo multi-insieme in cui viene inserito la variabile `r` usando la funzione `ins` della classe `MultiSet`, e da un resto `n` non inizializzato.

Capitolo 6

Un esempio: un sistema basato sulla riscrittura di multi-insiemi

In questo capitolo descriveremo alcuni semplici programmi d'esempio che mostrano in che modo si possono utilizzare i multi-insiemi e la programmazione con vincoli per risolvere alcuni problemi. In particolare mostreremo come realizzare un semplice interprete per il linguaggio GAMMA [9], il cui modello computazionale è basato sulla riscrittura di multi-insiemi, e come utilizzare questo interprete per risolvere due semplici problemi: il calcolo del fattoriale e la determinazione dei numeri primi.

6.1 L'interprete GAMMA in JSetL con multi-insiemi

GAMMA si propone di fornire un formalismo per scrivere programmi basati sulla nozione di parallelismo logico, ovvero la possibilità di definire un programma come semplice composizione di regole, senza preoccuparsi del con-

trollo imposto su di esse da un particolare parallelismo fisico. La caratteristica principale di un programma in GAMMA è la presenza di multi-insiemi i cui elementi possono interagire tra loro in base a determinate regole, il cui risultato è un nuovo multi-insieme. Un programma in GAMMA è composto da una collezione di regole non-deterministiche, ognuna delle quali è espressa attraverso vincoli su multi-insiemi, che se risolti generano un multi-insieme risultato, costruito a partire da quello iniziale, per trasformazioni successive.

Gli elementi fondamentali di un programma in GAMMA sono: R, ossia una funzione booleana che specifica se una determinata regola è applicabile; A che rappresenta l'azione di riscrittura di un multi-insieme su cui vale la regola R; l'operatore Γ che definisce il blocco fondamentale di un programma GAMMA. La valutazione di $\Gamma((R, A))(s)$, dove s è un multi-insieme, procede nel seguente modo: in modo non-deterministico viene selezionato, se esiste, un sottoinsieme di s dato da $\{\!\![x_1, \ldots, x_n]\!\!\}$; se $R(x_1, \ldots, x_n)$ restituisce `true` allora vengono eliminati tutti gli elementi $x_1, \ldots, x_n$ e viene eseguita l'azione $A(x_1, \ldots, x_n)$, aggiungendo eventualmente nuovi elementi al multi-insieme s; altrimenti se non esiste alcun sottoinsieme di s per cui valga la regola R, la computazione termina restituendo il valore corrente di s.

In CLP($\mathcal{B}ag$) l'implementazione dell'operatore Γ si ottiene definendo il predicato `gamma(M,M2)`, che stabilisce una relazione tra due multi-insiemi, quello iniziale M e quello finale $M2$, nel seguente modo:

```
gamma(Input,Input) :-
    not rule(Input,_).
gamma(Input,Output) :-
    rule(Input,Intermediate),
    gamma(Intermediate,Output).
```

Le regole di riscrittura dei multi-insiemi in GAMMA possono essere definite come regole di CLP($\mathcal{B}ag$) della forma generale:

`rule(InputMultiset,OutputMultiset)` :– *Condition.*

in cui `InputMultiset` e `OutputMultiset` sono multi-insiemi e *Condition* è la congiunzione di vincoli atomici che rappresentano l'azione A di cui si parlava precedentemente.

Osserviamo che la chiamata `rule(Input,_)` implica la risoluzione del vincolo di uguaglianza tra due multi-insiemi, ovvero `Input = InputMultiset`. Se `Input` è un multi-insieme del tipo $\{\!\!\{ t_1, \ldots, t_k \}\!\!\}$ e `InputMultiset` ha la forma $\{\!\!\{ x_1, \ldots, x_n | R \}\!\!\}$, con $n \leq k$, allora risolvere l'uguaglianza tra questi due multi-insiemi significa selezionare arbitrariamente un sottoinsieme $\{\!\!\{ x_1, \ldots, x_n \}\!\!\}$ del multi-insieme `Input`, come richiesto dalla definizione dell'operatore Γ.

La stessa tecnica usata con CLP($\mathcal{B}ag$) può essere utilizzata per realizzare un semplice interprete GAMMA con JSetL esteso con multi-insiemi:

```
public static void gamma(MultiSet M, MultiSet M2)
throws Failure{
   MultiSet M3 = new MultiSet();

   boolean res = rule(M, M3);
   if(res == false)
       Solver.add(M.eq(M2));
   else
       gamma(M3, M2);
   Solver.solve1();
  }
}
```

dove `rule` dovrà essere sostituita dalla chiamata alla funzione che realizza la specifica regola di riscrittura per un dato problema.

La funzione `gamma` prende in input due multi-insiemi `M` e `M2`, che rappresentano rispettivamente il multi-insieme `Input` e `Output` della definizione CLP($\mathcal{B}ag$). Il multi-insieme `M3` è quello intermedio, cioè il multi-insieme `Intermediate` utilizzato in input dalla stessa `gamma`, che viene richiamata se la `rule` è applicabile. Questo ultimo concetto è stato tradotto in un controllo `if then else` sul valore booleano `res`, che contiene il risultato della regola `rule` applicata ai multi-insiemi `M` ed `M3`. Alla fine la procedura `gamma` richiama la `solve1()`, funzione della classe `SolverClass` che funziona come la `solve`, ma restituisce soltanto la prima soluzione dell'insieme dei vincoli contenuti nel constraint store e rimuove dallo store tutte le alternative rimaste inesplorate.

Vediamo come GAMMA può essere usato per risolvere due semplici problemi matematici.

6.2 Il fattoriale in GAMMA

Il problema è descritto come segue: dato un numero n si richiede il calcolo del fattoriale $n!$. Per definizione sappiamo che:

$$n! = \begin{cases} 1 & se\ n = 0 \\ n(n-1)! & altrimenti \end{cases}$$

Per modellare il problema utilizziamo un interprete GAMMA. Il multi-insieme iniziale sarà il multi-insieme costituito dai numeri da 1 a n (vuoto se $n = 0$);

il multi-insieme finale conterrà un solo elemento , $n!$. In questo caso la regola R che vogliamo applicare è `factorialRule`, che calcola il fattoriale di un numero n. La definizione CLP($\mathcal{B}ag$) di questa regola è:

```
factorialRule({{X,Y|R}}, {{Z,R}}):-
                Z is X * Y.
factorialRule({{}}, {{1}}).
```

Analizziamo ora la realizzazione della `factorialRule`, in JSetL:

```
public static boolean factorialRule(MultiSet M, MultiSet M2)
throws Failure{
  Lvar x = new Lvar();
  Lvar y = new Lvar();
  Lvar z = new Lvar();
  MultiSet R = new MultiSet();

  MultiSet Cfr = MultiSet.empty.ins(new Lvar());
  if (Solver.boolSolve1(M.eq(Cfr)))
      return false;

  if(M.isEmpty())
      Solver.add(M2.eq(MultiSet.empty.ins(1)));

  else{
  Solver.add(M.eq(R.ins(x).ins(y)));
  Solver.add(z.eq(x.mul(y)));
  Solver.add(M2.eq(R.ins(z)));
  }
```

```
return Solver.boolSolve1();

}
```

Nel multi-insieme M sono memorizzati tutti gli interi compresi tra 1 e il numero n di cui si vuole calcolare il fattoriale; il multi-insieme $M2$ conterrà il risultato finale. La funzione `factorialRule` restituisce un booleano, risultato della `boolSolve1()`, che restituisce `true` se la `solve1()` riesce a trovare una soluzione per tutti i vincoli memorizzati nello store, altrimenti restituisce `false`.

Per calcolare il fattoriale si impongono i seguenti vincoli: $M.eq(\{\!\!\{ x, y|R \}\!\!\})$, $z.eq(x.mul(y))$ e $M2.eq(\{\!\!\{ z|R \}\!\!\})$, ovvero prendendo due elementi x, y di M, si calcola il loro prodotto attraverso il vincolo mul e lo si memorizza in una variabile nuova z. Il resto R contiene tutti gli elementi che non sono stati moltiplicati tra loro; uguagliando $M2$ e $\{\!\!\{ z|R \}\!\!\}$ in $M2$ vengono memorizzati gli stessi elementi di R. Alla fine il multi-insieme $M2$ conterrà soltanto un valore z, prodotto di tutti gli elementi di M, ovvero il fattoriale $n!$. Non appena l'interprete GAMMA richiamerà la `factorialRule` su un multi-insieme con un solo elemento, la regola uscirà subito con un `false` e quindi il multi-insieme stesso viene restituito come risultato calcolato dall'interprete GAMMA.

La funzione `main` di questo esempio si occupa di interagire con l'utente per sapere di quale numero si deve calcolare il fattoriale, di creare il multi-insieme M, richiamare l'interprete `gamma` e stampare il risultato:

```
public static void main(String[] args)
throws IOException, NumberFormatException, Failure {
  System.out.println("Calcolo del fattoriale\n");
  System.out.println("Dammi il numero ");
  InputStreamReader reader = new InputStreamReader(System.in);
  BufferedReader input = new BufferedReader(reader);
  String number = input.readLine();
  int n = Integer.parseInt(number);

  MultiSet S = new MultiSet();
  MultiSet M = new MultiSet("M",MultiSet.empty);

  for (int i = 1; i <= n; i++)
      M = (M.ins(i));

  gamma(M,S);
  System.out.println("\nIl fattoriale di "+n+ " è " );
  S.output();
}
```

Eseguendo questo programma con il valore $n = 5$ abbiamo il seguente output:

```
Calcolo del fattoriale
Dammi il numero di cui vuoi calcolare il fattoriale
5
Il fattoriale di 5 è

MultiSet_1 = +{120}+
```

Ecco come procede: il ciclo `for` del `main` crea un multi-insieme M contenente tutti gli interi compresi tra 1 e 5 ovvero $M = \{\!\!\{ 4, 3, 2, 1 \}\!\!\}$. Poi richiama la procedura `gamma(M,S)` dove `S` è il multi-insieme che conterrà il risultato e che rappresenta il secondo parametro della funzione `gamma`. Alla prima chiamata della `factorialRule` viene creato il nuovo multi-insieme $M3$ che rappresenta il parametro `Intermediate`. Vengono risolti i vincoli nel constraint store nel seguente modo:

```
M = +{4,3,2,1}+
x = 3
y = 4
z = 12
M3 = +{12,2,1}+
R = +{2,1}+
```

ovvero vengono selezionati in modo non-deterministico 3 e 4 il cui prodotto viene memorizzato in z. Il resto R diventa il multi-insieme composto da $1, 2$. Ora l'ultimo vincolo di uguaglianza $M2.eq(R.ins(z))$ mi permette di scrivere il multi-insieme intermedio $M3$ contenente il prodotto eseguito in questa prima iterazione della `factorialRule` e il resto degli elementi non ancora moltiplicati. Al passaggio successivo viene richiamata la `factorialRule` sul multi-insieme intermedio precedentemente creato, ovvero $M3 = \{\!\!\{ 12, 2, 1 \}\!\!\}$. La risoluzione dei vincoli ha ancora successo e la computazione procede nel seguente modo:

```
M3 = +{12,2,1}+
x = 2
y = 12
```

```
z = 24
M3 = +{24,1}+
R = +{1}+
```

Dal multi-insieme $M3$ vengono selezionati ora gli interi 2 e 12 che moltiplicati restituiscono il valore 24 che sarà memorizzato prima in z, poi in un nuovo multi-insieme intermedio assieme al resto R contenente solo il valore 1, l'ultimo numero che deve essere moltiplicato. Poi il multi-insieme $M3 = \{\!\!\{ 24, 1 \}\!\!\}$ viene passato come primo parametro della prossima chiamata alla `factorialRule`, la quale restituisce ancora `true` e risolve i vincoli nel seguente modo:

```
M3 = +{24,1}+
x = 1
y = 24
z = 24
M3 = +{24}+
R = +{}+
```

Dopo aver moltiplicato gli ultimi due numeri presenti nel multi-insieme in ingresso e memorizzato il risultato, ovvero il valore 24 nel multi-insieme intermedio creato in questa iterazione, il resto R risulta vuoto. Quando viene richiamata la funzione `gamma` sul multi-insieme contenente un solo elemento, la `boolSolve1()` della `factorialRule` restituisce `false`, quindi la procedura `gamma` aggiunge al constraint store un nuovo vincolo che risolto restituisce l'ultimo multi-insieme creato dalla `factorialRule`, cioè $M3 = \{\!\!\{ 24 \}\!\!\}$.

6.3 I numeri primi in GAMMA

Si consideri il seguente problema: dato un numero n vogliamo determinare tutti i numeri primi minori del numero n. La soluzione tramite GAMMA prevede di dare in input all'interprete GAMMA un multi-insieme contenente tutti gli interi da 2 a n e l'interprete restituisce un multi-insieme contenente soltanto gli interi del multi-insieme di partenza che sono numeri primi.

L'interprete GAMMA è lo stesso usato per il calcolo del fattoriale, con l'unica differenza che in questo problema la regola da applicare sarà `primeRule`. Per il resto il codice della procedura `gamma` non è cambiato, quindi possiamo dire che essa è in un certo senso parametrica rispetto alla regola da applicare ai multi-insiemi in ingresso. Infatti in questo caso la regola R che vogliamo applicare è `primeRule`, il cui risultato viene memorizzato nel valore booleano `res`:

```
boolean res = primeRule(M, M3);
```

I due parametri in input sono sempre multi-insiemi, il primo contenente tutti gli interi compresi tra 2 e il numero n passato dall'utente, mentre il secondo, non inizializzato, conterrà il risultato di ogni iterazione del metodo `primeRule`. La `primeRule` è implementata come segue:

```
public static boolean primeRule(MultiSet M, MultiSet M2)
throws Failure{

        boolean res = true;
        Lvar x = new Lvar();
        Lvar y = new Lvar();
```

```
        Lvar z = new Lvar();
        MultiSet R = new MultiSet();

        Solver.add(M.eq(R.ins(x)));
        Solver.add(y.in(R));
        Solver.add(z.eq(x.mod(y)).and(z.eq(0)));
        Solver.add(M2.eq(R));

        res = Solver.boolSolve1();

        if (!M2.known()) {
            Solver.clearStore();
            res = false;
        }

        return res;
    }
```

Il meccanismo di risoluzione di questo problema è del tutto analogo a quello del problema del fattoriale, ovvero la procedura `gamma` richiama la funzione `primeRule` su trasformazioni successive del multi-insieme iniziale, restituendo `true` finchè l'insieme dei vincoli ha soluzione; nel momento in cui questo non è più possibile, la `boolSolve1()` ritorna `false` e la procedura `gamma` restituisce l'ultimo multi-insieme creato.

In questo esempio la condizione di uscita dalla funzione `primeRule` è data dal ramo `if` in cui si controlla che il multi-insieme creato ad ogni iterazione non sia completamente noto, attraverso la negazione della funzione `known()`, applicata al multi-insieme $M2$. In questo caso significa che in $M2$ in realtà

è già presente l'insieme dei numeri primi minori di n, ovvero il risultato; allora dobbiamo eliminare tutti i vincoli che sono stati introdotti nello store da quest'ultima iterazione, in quanto inutili, dobbiamo quindi restituire un `false`. Ora la procedura `gamma` entrerà nel ramo `else` e aggiungerà il solo vincolo di uguaglianza tra i multi-insiemi M ed $M2$. Risolvendo si trova il risultato atteso.

Analizziamo i vincoli che vengono imposti per il calcolo dei numeri primi: $M.eq(R.ins(x))$, ovvero unifichiamo il multi-insieme in input M con il multi-insieme $\{\!|\, x|R \,|\!\}$ dove nè x variabile logica, nè R multi-insieme sono inizializzati. Poi introduciamo $y.in(R)$, $z.eq(x.mod(y))$ e $z.eq(0)$, ovvero prendiamo un elemento nel resto R in modo non-deterministico, calcoliamo il resto di x modulo y, attraverso il vincolo mod memorizzandolo in z che deve essere 0. Questo significa che gli interi considerati x ed y sono uno multiplo dell'altro. In tal caso creiamo il multi-insieme risultato $M2$ inserendo solo tutti gli elementi di R, cioè eliminando dal multi-insieme di partenza il valore x, che risulta essere un multiplo di y e quindi non potrà mai essere un numero primo.

Il corpo della funzione `main` non ha subìto cambiamenti rilevanti rispetto all'esempio precedente, se non per i messaggi di output e per la costruzione del multi-insieme M che conterrà ora tutti gli interi compresi tra 2 e n e non più da 1 ad n, infatti eseguendo il programma per il calcolo dei numeri primi con l'intero 9 abbiamo il seguente output:

```
Calcolo dei numeri primi
Dammi un numero intero
9
```

```
I numeri primi minori di 9 sono
MultiSet_1 = +{7,5,3,2}+
```

che è il risultato che ci aspettavamo. Vediamo come procede il programma stampando i valori delle variabili e dei multi-insiemi creati ad ogni iterazione della procedura `gamma` e quindi della funzione `primeRule`. Inizialmente si ha:

```
M = +{9,8,7,6,5,4,3,2}+
x = 9
R = +{8,7,6,5,4,3,2}+
y = 3
M2 = +{8,7,6,5,4,3,2}+
res: true
```

M è il multi-insieme creato dal ciclo `for` del `main` dopo aver memorizzato l'intero n e rappresenta il primo parametro della funzione `primeRule`. Viene scelto $x = 9$ e dal resto R viene selezionato $y = 3$ perchè deve valere il vincolo $(x.mod(y)).eq(0)$, allora viene creato il multi-insieme $M2$ eliminando il valore 9 da M, risolvendo il vincolo $M2.eq(R)$. La `boolSolve1()` ha restituito `true` quindi la procedura `gamma` richiama la funzione `primeRule`, passando come primo parametro il multi-insieme $M2$ appena creato. Risolvendo ancora l'insieme dei vincoli si ottiene:

```
M = +{8,7,6,5,4,3,2}+
x = 8
R = +{7,6,5,4,3,2}+
y = 4
M2 = +{7,6,5,4,3,2}+
```

```
res: true
```

Ora viene selezionato $x = 8$, quindi il resto R contiene gli elementi compresi tra 7 e 2, tra essi viene scelto $y = 4$ sempre per poter soddisfare il vincolo $(x.mod(y)).eq(0)$, viene creato $M2$ uguale ad R e il risultato booleano è ancora `true`. Alla prossima iterazione, partendo dal multi-insieme $M = \{\!\!\{ 7, 6, 5, 4, 3, 2 \}\!\!\}$ si settano le variabili nel seguente modo:

```
M = +{7,6,5,4,3,2}+
x = 6
R = +{7,5,4,3,2}+
y = 3
M2 = +{7,5,4,3,2}+
res: true
```

Questa volta gli interi su cui verrrà eseguita l'operazione di modulo imponendo che il resto sia 0 sono $x = 6$ e $y = 3$, quindi il nuovo multi-insieme è $M2 = \{\!\!\{ 7, 5, 4, 3, 2 \}\!\!\}$ e il valore booleano passato alla procedura `gamma` è di nuovo `true`. Quindi verrà eseguita un'altra iterazione della `primeRule`, in cui le variabili avranno i seguenti valori:

```
M = +{7,5,4,3,2}+
x = 4
R = +{7,5,3,2}+
y = 2
M2 = +{7,5,3,2}+
res: true
```

Il vincolo $(x.mod(y)).eq(0)$ viene risolto nel seguente modo: $(4.mod(2)).eq(0)$, allora il nuovo multi-insieme $M2$ uguale ad R conterrà i valori $7, 5, 3, 2$ e la variabile booleana `res` avrà valore `true`. Quindi verrà eseguita un'iterazione in più della funzione `primeRule`, in quanto il multi-insieme $M2$ è il risultato corretto. A questa iterazione accade che l'insieme dei vincoli nel constraint store non ha soluzione e il multi-insieme $M2$, secondo parametro della `priemRule`, non può essere inizializzato, quindi il controllo `if(!M2.known())` ha successo, allora viene ripulito lo store e la funzione restituisce `false`. A questo punto la procedura `gamma` ritorna l'ultimo multi-insieme creato, ovvero $M2 = \{\!\!\{ 7, 5, 3, 2 \}\!\!\}$ che è il risultato corretto.

Appendice A

Manual for Multiset

A.1 Multiset: the class MultiSet

A Multiset is a data structure, similar to a set, except that the number of the occurrences of the same element is important. In JSetL a multiset is defined as an instance of the class `MultiSet`. Its value is a finite, possibly empty, collection of arbitrary `Lvar-`*`values`*, name as elements of the multiset. A multiset with name `MSetName` can be created in JSetL by the Java statement (a *multiset declaration*)

```
MultiSet MSetName = new MultiSet(MSetNameExt,MSetElemValues);
```

where `MSetNameExt` is an optional *external name* of the multiset (the default external name has the form "`MultiSet_n`" where `n` is a unique integer), and `MSetElemValues` is an optional part which is used to specify the elements of the multiset when it is created (see below).
The *empty multiset* is denoted by the constant `MultiSet.empty`.
Multisets can be created through the **new** operator, possibly using multiset declarations, or as the result of executing utility methods deling with multi-

sets (see next sections). Multisets, like sets and the other data structures in JSetL, can be either initialized or uninitialized. The value of a multiset (i.e. a `MultiSet-`*`value`*) is the collection of its elements. In particular, when a multiset is created through the **new** operator, the value of the multiset can be specified in exactly the same ways seen for sets and lists: by passing an array of its elements, or a `MultiSet` or a *multiset*-`MultiSet` object.
The notation we will use to write multiset is similar to the one used for sets, apart using a double pair of curly brackets instead of a single pair of the curly brackets. For instance, the multiset containing the elements *a*,*b*, and two occurrences of *c* will represented as $\{\{a,b,c,c\}\}$, and the empty multiset will be represented as $\{\{\}\}$. Moreover,

$$\{\{e_1, e_2, \ldots, e_n | S\}\},$$

where S is a possibly uninitialized multiset, is used to denote a multiset containing elements e_1,e_2,...,e_n, plus elements in S.

Esempio 19 . ***MultiSet*** *definitions*

```
MultiSet m = MultiSet.empty();        //the empty multiset
MultiSet n = new MultiSet("n");//an uninitialized multiset,
                                        with ext'l name "n"
int[] v_elems = {2,2,3,2,4};
MultiSet v = new MultiSet("v",v_elems);    //an inizialized
                               multiset, with ext'l name "v"
                               and value {2,2,3,2,4}
```

Note that the main difference between sets and multisets is the importance of the repetition of the same element. In fact, for example, the

`Multiset-`*`values`* {1,2}, {1,2,2} and {1,2,1} all represent the same set, but they denote three different multisets.

Elements of a multiset can be also logical variables (or lists or sets) possibly uninitialized. Multisets that contain uninitialized elements are said to be partially specified multisets. Like the other data structures of JSetL, multisets can also be nested, at any depth.

Esempio 20 . *Partially specified and nested multisets*

```
Lvar x = new Lvar();
Object[] ps_elem = {new Integer(1),x};

MultiSet ps = new MultiSet(ps_elem);          //the partially
                                    specified multiset {{1,x}}
MultiSet m = MultiSet.empty;
MultiSet n = new MultiSet("n");

int[] v_elems = {2,2,3,2,4};
MultiSet v = new MultiSet("v",v_elems);

MultiSet[] ns_elems = {m,n,v};          //m,n,v are multisets
MultiSet ns = new MultiSet("ns",ns_elems);      //a multiset
                                   containing nested multisets
```

Note that, differently from sets, the cardinality of a partially specified multiset is determined precisely. For example, the cardinality of the multiset `ps` of Example 2, is surely 2, either x will be instantiated to a value equal to 1 or different from 1.

Multisets are manipulated through constraints. Besides constraints, in JSetL there are also element insertion operations and a limited number of utility methods, that allow to work with multisets.
In the next section, we will descibe the insertion operations, which are fondamental for the definition of multises, and the other utility methods.

A.2 Multiset element insertion

JSetL provides two methods for inserting elements in a multiset: `ins`, for the insertion of a single element, and `insAll` for the insertion of an array of elements. Like sets, since the order of elements in a multiset is not important, it is not necessary to provide distinct methods for head and tail insertion because they would produce the same multiset.
`ins` and `insAll` can be invoked on any kind of multiset (namely by the user or by the JSetL, completely or partially specified), with no restriction on the presence of unintialized logical variables in their arguments. Both methods do not modify the multiset on which they are invoked: rather they build and return a new multiset obtained by adding the elements to the multiset:

```
public MultiSet ins(t elem);
```

where `t` *represent a* `Lvar`*-value type.*

```
MultiSet s = new MultiSet("s");
s.ins(e);
```

returns a multiset obtained by adding `e` *to the multiset* `s`

```
public MultiSet insAll(t[] elem_array);
s.insAll(a);
```

where `a` *is an array of elements of type* `t`, *returns a multiset by adding all elements of* `a` *to the multiset* `s`.

The `ins` and `insAll` methods can be concatenated (left associated). In fact these methods always return a multiset object, and the returned object can be used as the invocation object as well.

Esempio 21 . *Multiset element insertion*

```
Lvar nil = new Lvar(MultiSet.empty);    //the empty multiset

MultiSet m1 = new MultiSet(nil.ins(1)); //the multiset {{1}}

Lvar x = new Lvar();
MultiSet m2 = new MultiSet(m1.ins(2).ins(x));
                                        //the multiset {{2,x}}

int[] s_elems = {2,1,2,3};
MultiSet m3 = new MultiSet(nil.insAll(s_elems));
                                    //the multiset {{2,1,2,3}}

MultiSet r = new MultiSet();     //an uninitialized multiset
MultiSet m4 = new MultiSet(r.ins(1));        //the unbounded
                                            multiset {{1|r}}
```

Note that using the insertion methods `ins` and `insAll` it is possible to build unbounded partially specified multisets, that is multisets with a certain number of elements (either known or unknown) and a *"rest"* of the multiset, represented by an uninitialized multiset `r` (i.e. using the abstract notation $\{\{e_1,e_2,\ldots,e_n|\ \mathrm{r}\}\}$).

A.3 Multiset constraints

Let us analyze some particular cases involving atomic constraints based on multiset-theoretical operations (i.e *multiset constraints*).

In all these constraints, expressions `s`,s_1,s_2 and s_3 can be *uninitialized* `Lvar` or `Multiset` objects. The constraint solver is always able to solve all these cases, leaving the solved form constraint in the constraint store or generating a `Failure` exception.
As an example, let us consider a membership constraint `x` $\in$ `s`, where is an uninitialized multiset, to be added to the constraint store:

```
Solver.add(x.in(s));
```

The constraint solver rewrites this constraint to the equality constraint

```
s = {x|N}
```

where `N` *is generated uninitialized Multiset object, which states, in terms of equality, that* `s` *must contain* `x` *plus something else, denoted by* `N`.
Like set the `in` and `nin` multiset constraints can be invoked also on uninitialized multisets with unknown elements. Thus, for instance, the `in` constraint can be used to obtain nondeterministically all elements of a multiset, or to build a multiset that contains a given element. Multisets involved in multiset constraints can also be partially specified, either bounded or unbounded.

Esempio 22 . *Constraint solving*

```
MultiSet sr = new MultiSet();   //an uninizialized multiset
Lvar X = new Lvar();
Lvar Y = new Lvar();
Lvar Z = new Lvar();
```

```
int[] a = {1,1,2};

MultiSet s = new MultiSet(sr.ins(X).ins(Y).ins(Z));
                                          //s is {{X,Y,Z|sr}}

MultiSet r = new MultiSet(sr.insAll(a));//r is {{1,1,2|sr}}

Solver.add(r.eq(s));        // multiset unification r = s
Solver.add(X.neq(1));       // X != 1
Solver.solve();             //calling the constraint solver
X.output();
```

The output generated by this code fragment is:

```
X = 2
```

The value for `X` *is computed through backtracking. Solving the multiset unification problem* `{X,Y,Z} = {1,1,2}` *nondeterministically returns one of the different solutions below:*

```
X = 1, Y = 1, Z = 2,
X = 1, Y = 2, Z = 1,
X = 2, Y = 1, Z = 1.
```

Assuming the first value for `X` *is 1, then the other constraint,* `x.neq(1)`*,turns out to be not satisfied. Thus, backtracking forces the solver to find another solution for* `X`*,namely* `X = 2`*. In this case, the conjunction of two given constraints is satisfied, and the invocation of the* `solve` *method terminates successfully.*

Bibliografia

[1] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo.
JSetL: A Java Library for Supporting Declarative Progamming in Java
In corso di pubblicazione su *Software-Practice & Experience*,
ISSN: 0038-0644, 2006.

[2] Elio Panegai, Elisabetta Poleo, Gianfranco Rossi.
JSetL User's Manual - Versione 1.0
Research Report Quaderno del Dipartimento di Matematica, n 384, Università degli Studi di Parma, November 2004.

[3] Agostino Dovier, Carla Piazza, Gianfranco Rossi.
A uniform approch to constraint-solving for lists, multisets, compact lists, and sets.
Under revision, 2005.

[4] A. Dovier, A. Policriti, G. Rossi.
A uniform axiomatic view of lists, multisets, and sets and the relevant unification algorithm
Fundamenta Informaticae, 1998.

[5] Agostino Dovier, Carla Piazza, Gianfranco Rossi.
Multiset rewriting by multiset constraint solving. Romanian Journal of Information Science and Technology, Vol. 4(1–2): 59–76, 2001.

[6] Agostino Dovier, Carla Piazza, Enrico Pontelli, Gianfranco Rossi.
Set and Constraint Logic Programming.
ACM Toplas, 22(5) 2000, 861-931.

[7] Gianfranco Rossi, Elisabetta Poleo.
JSetL: Declarative Programming in Java with Sets
In *C.F. '04, 2004 ACM International Conference on Computing Frontiers,* ACM Press, ISBN/ISSN: 1-58113-741-9,2004.

[8] P. Arenas-Sanchez, F.J. Lopez-Fraguas, M.Rodriguez-Artalejo.
Embedding Multiset Constraints into a Lazy Functional Logic Language.
In *Proc. Programming Languages Implementation and Logic Programming (PLILP'94)*
LNCS 1490: 429 - 444. Springer-Verlag, 1998.

[9] J.P Banâtre, D. Le Métayer.
Programming by Multiset Transformation.
In *Comunications of the ACM,* 36(1): 98 - 111, January 1993.

[10] J. Jaffer, M. J. Maher.
Constraint Logic Programming: a Survey.
In *Journal Logic Programming* 19, 20: 503 - 581, 1994.

[11] SICStus PROLOG User's Manual.
Swedish Institute of Computer Science, 1991.

[12] G. Berry and G. Boudol.
The Chemical Abstract Machine.
Theoretical Computer Science vol. 96 (1992) 217-248.

[13] G. Paun.

Computing with Membranes.

Journal of Computer and System Science, 61, 2000.

Ringraziamenti

E' doveroso, a questo punto della mia carriera scolastica, dedicare qualche riga alle persone che mi hanno accompagnata in questa esperienza e il cui contributo è stato essenziale.
Innanzi tutto devo ringraziare le persone senza le quali non avrei mai raggiunto un simile traguardo: i miei genitori, che sono stati, in tutti questi anni, i miei sostenitori morali e materiali; loro mi hanno guidata, consigliata, spronata e consolata quando ne avevo bisogno. A loro, che non hanno mai smesso di essere orgogliosi di me e di avere fiducia nelle mie capacità, dedico il mio successo scolastico più grande.

Devo inoltre ringraziare il mio relatore il Prof. Gianfranco Rossi ed il mio correlatore il Dott. Elio Panegai, che mi hanno accompagnata nei mesi di tirocinio e in questo lavoro di tesi. Il confronto con loro è stato entusiasmante e produttivo, sia dal punto di vista didattico che umano, infatti lavorare al loro fianco è stato per me fonte di arricchimento personale e non solo per la possibilità di sfruttare la loro esperienza professionale. Sono sempre stati disponibili, nonostante gli innumerevoli impegni, per chiarire ogni mio dubbio e perplessità. Grazie a loro ho potuto partecipare attivamente ad una squadra di progetto, dando un senso diverso alle ore passate sul computer a programmare, non in funzione di un esame o per svolgere un esercizio che sarà dimenticato in un cassetto, ma per apportare modifiche ad un progetto reale.

Il Prof. Rossi ha saputo essere in questi anni un docente eccezionale; tutte le regole difficili, spiegate da lui, attraverso quei buffi esempi, diventavano non solo semplici e comprensibili in un attimo, ma quasi divertenti. La sua tranquillità e pacatezza riuscivano sempre a fare sentire chiunque a suo agio, sia a ricevimento, quando si facevano consapevolmente domande sciocche, che in sede d'esame.

Voglio ringraziare le mie sorelle, la mia più cara amica e compagna di avventure Daniela, i miei famigliari e tutti gli amici che mi hanno seguita con stima ed interesse nella scalata verso la laurea. Un altro grazie va a tutti quei pazzi, matematici ed informatici, che davvero possono capire ogni discorso strano e senza senso per tutti i comuni mortali, fatto tra queste e quelle mura. Infine, l'ultimo ringraziamento va a Beppe, non per importanza, ma solo perchè è la persona che per ultima in ordine di tempo è entrata nella mia vita. Lui in questi mesi mi è stato vicino in un discreto silenzio, sopportando i miei sfoghi e le giornate cariche del mio solito pessimismo.

Dimenticavo ..., non posso non ringraziare tutte le persone, sia dell'ambiente scolastico che del mio quotidiano, che nel corso di questi anni hanno cercato di dissuadermi nel perseguire il mio obiettivo e che non hanno mai creduto nelle mie capacità. A loro dico grazie, perchè l'orgoglio e l'autostima che hanno cercato di scalfire, sono invece cresciute dentro di me e mi hanno spinta con grinta e determinazione fino alla tanto irraggiungibile laurea.

A tutte queste persone grazie!!!

Nadia