1

l

UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE

MATEMATICHE, FISICHE e NATURALI

Corso di Laurea in Informatica

Tesi di Laurea Triennale

Trattamento uniforme di insiemi, multi-insiemi e liste nella libreria JSetL

Candidato: **Luca Pedrelli**

Relatore: **Prof. Gianfranco Rossi**

Anno Accademico 2007/2008

Dedicato alla mia famiglia
alla mia ragazza
e a tutti quelli che mi hanno aiutato

Indice

Introduzione

JSetL [5] è una libreria Java [7] che combina il paradigma di Java della programmazione orientata agli oggetti con importanti concetti dei linguaggi CLP (Constraint Logic Programming), come variabili logiche, liste (eventualmente parzialmente specificate), unificazione, constraint solving, non-determinismo. La libreria prevede anche insiemi e vincoli su insiemi come quelli presenti in CLP($\mathcal{SET}$) [2] e, in modo molto limitato, anche multi-insiemi e vincoli sui multi-insiemi come quelli descritti in [3]. L'unificazione può coinvolgere variabili logiche, oppure oggetti di tipo lista o insieme ("set unification") o multi-insieme. I vincoli riguardano operazioni della teoria base degli insiemi (per esempio appartnenza, unione, intersezione, etc.), ma anche operatori di confronto (uguaglianza, disuguaglianza) tra interi. Il constraint solving in JSetL è caratterizzato da "punti di scelta" e "backtraking". In particolare, le soluzioni per vincoli su insiemi sono calcolate in modo non-deterministico, usando procedure non-deterministiche di riscrittura dei vincoli secondo la tecnica definita in [2].

In JSetL le variabili logiche hanno molte caratteristiche in comune con gli insiemi, i multi-insiemi e le liste, ad esempio l'utilizzo dei vincoli di uguaglianza, disuguaglianza, appartenenza e non-appartenenza. Nonostante questo, la libreria usa solo in rari casi l'ereditarietà e non vi sono interfacce che possano offrire all'utente o al package una gestione uniforme. Un altro aspetto da considerare è quello relativo alle caratteristiche comuni tra insiemi, multi-insiemi e liste. Queste caratteristiche sono date da alcune proprietà e

dall'utilizzo di vincoli come unione, disgiunzione, intersezione e differenza. Anche in questo caso la versione attuale della libreria non usa l'ereditarietà. Riguardo la gestione dei vincoli su liste e multi-insiemi, i vincoli implementati sono molto pochi e alcuni aspetti teorici relativi alla risoluzione dei vincoli in queste strutture dati non sono trattati.

Un obbiettivo di questa tesi è quello di costruire una o più interfacce che permettano, a livello utente (interfacce pubbliche) o internamente al package (interfacce protected), di unire i concetti comuni fra variabili logiche, insiemi, liste e multi-insiemi, mantenendo la compatibilità con JSetL 1.3.2. Questo argomento verrà trattato nei Capitoli 1 e 2. Le intefacce implementate per il raggiungimento di questo obbiettivo sono descritte in dettaglio in appendice A.
Un altro obbiettivo della tesi è quello di ampliare il trattamento dei multi-insiemi e dei vincoli su multi-insiemi all'interno della nuova libreria (JSetL 1.3.3). Per questo verrà data la definizione formale di multi-insieme e delle operazioni di appartenenza multipla, unione somma, disgiunzione, sottoinsieme, intersezione, differenza e unione massimo. Per ognuna di queste operazioni verrà dimostrato che esiste una congiunzione di operazioni, composte solo da unione somma e disgiunzione, equivalente. In base ai risultati teorici verranno definite le regole di riscrittura dei vincoli di appartenenza multipla, unione somma, disgiunzione, intersezione, differenza e unione massimo. Questi argomenti verranno trattati nel Capitolo 3. Nel Capitolo 4 verrà mostrata l'implementazione Java del trattamento dei vincoli su multi-insiemi sviluppato nel Capitolo 3.

Capitolo 1

Insiemi, Liste e Multi-insiemi in JSetL 1.3.2

JSetL è una libreria che unisce la programazione a oggetti di Java con alcune funzionalità per supportare la programmazione dichiarativa, tra cui variabili logiche, insiemi, liste, multi-insiemi, non-determinismo e constraint solving, tipiche dei linguaggi di programmazione logica a vincoli. In questo Capitolo ci si sofferma sulla gestione delle strutture dati principali di JSetL come le variabili logiche, gli insiemi, i multi-insiemi e le liste. È infatti importante, per gli scopi di questa tesi, aver presente come sono rappresentate queste quattro strutture all'interno della libreria, per poter comprendere e sviluppare le idee che hanno portato alla nuova architettura di JSetL 1.3.3.

1.1 La nozione di variabile logica

JSetL [6] supporta la nozione di **variabile logica**, solitamente fornita dai linguaggi di programmazione logica e funzionale. Una variabile logica può essere **inizializzata** o **non-inizializzata**. Il valore di una variabile logica in JSetL può essere di qualunque tipo e può essere specificato alla creazione o come risultato di elaborazione di vincoli che la coinvolgono, in particolare, vincoli di uguaglianza. I vincoli possono essere usati sia per "settare" sia

per controllare il valore di una variabile logica, ma non per modificarlo.

Definizione 1.1 *Una **variabile logica** è un'istanza della classe* `Lvar`*, creata dalla dichiarazione*

```
Lvar nameVar = new Lvar(NameVarExt, ValueVar);
```

dove `nameVar` è il nome dell'oggetto Java, `NameVarExt` (parametro opzionale) è un nome esterno, `ValueVar` (parametro opzionale) è un valore di inizializzazione della variabile stessa.

Definizione 1.2 *Una **variabile logica** che non ha un valore associato ad essa è detta **non-inizializzata** (o incognita). Altrimenti la variabile è detta **inizializzata**.*

Oltre ai vincoli, la classe `Lvar` fornisce metodi che permettono di leggere e scrivere il valore della variabile logica, di conoscere se la variabile è inizializzata o no, di ottenere il suo nome esterno e così via.

1.2 La nozione di insieme

Definizione 1.3 *Un **insieme logico** in JSetL è un'istanza della classe* `LSet`*, creata dalla dichiarazione*

```
LSet nameSet = new ConcreteLSet(NameSetExt, ValueSetElem);
```

dove `nameSet` è il nome dell'oggetto Java, `NameSetExt` (parametro opzionale) è un nome esterno, `ValueSetElem` (parametro opzionale) è usato per specificare gli elementi dell'insieme e può essere un array di elementi di qualsiasi tipo, o i limiti `l` e `u` di tipo `int` di un intervallo [`l`,`u`] di numeri interi. Se `ValueSetElem` non è specificato viene costruito un insieme non inizializzato. L'insieme vuoto è denotato dalla costante `ConcreteLSet.empty`. Le dichiarazioni hanno quindi la forma (con `arr` array di un qualsiasi tipo primitivo, `l` e `u` di tipo primitivo `int`)

```
LSet nameSet = new ConcreteLSet();
LSet nameSet = new ConcreteLSet(NameSetExt);
LSet nameSet = new ConcreteLSet(arr);
LSet nameSet = new ConcreteLSet(NameSetExt,arr);
LSet nameSet = new ConcreteLSet(NameSetExt,l,u);
LSet nameSet = ConcreteLSet.empty;
```

Nella prima dichiarazione `nameSet` è un insieme **non-inizializzato** con nome esterno non specificato.
Nella seconda dichiarazione `nameSet` è un insieme **non-inizializzato** con nome esterno `NameSetExt`.
Nella terza dichiarazione `nameSet` è un insieme con nome non specificato e con elementi `arr`.
Nella quarta dichiarazione `nameSet` è un insieme con nome esterno `NameSetExt` e con elementi `arr`.
Nella quinta dichiarazione `nameSet` è un insieme con nome esterno `NameSetExt` che ha per elementi i numeri interi dell'intervallo [`l`,`u`].
Nella sesta dichiarazione `nameSet` è l'insieme vuoto.

Vediamo alcuni esempi più specifici sulla definizione di insiemi logici:

```
int[] elements = {1,3,10};
LSet S = new ConreteLSet("S", elements);
LSet R = new ConreteLSet();
LSet Y = R.insAll(elements);
```

Nella prima dichiarazione si istanzia un array di elementi interi chiamato `elements`. Nella seconda si definisce l'insieme $S = \{1, 3, 10\}$ di nome `S`. Nella terza si definisce l'insieme **non-bounded** $Y = \{1, 3, 10|R\}$, con elementi specificati `elements` ed il resto degli elementi non specificati.

1.3 La nozione di lista

Definizione 1.4 *Una **lista logica** in JSetL è un'istanza della classe* `Lst`*, creata dalla dichiarazione*

```
Lst nameLst = new Lst(NameLstExt, ValueLstElem);
```

dove `nameLst` (parametro opzionale) è il nome dell'oggetto Java, `NameLstExt` (parametro opzionale) è un nome esterno, `ValueLstElem` è un parametro opzionale usato per specificare gli elementi della lista e può essere un array di elementi di qualsiasi tipo. La lista vuota è denotata dalla costante `Lst.empty`. Le dichiarazioni hanno quindi la forma (con `arr` array di un qualsiasi tipo primitivo)

```
Lst nameLst = new Lst();
Lst nameLst = new Lst(NameLstExt);
Lst nameLst = new Lst(arr);
Lst nameLst = new Lst(NameLstExt,arr);
Lst nameLst = Lst.empty;
```

Nella prima dichiarazione `nameLst` è una lista **non-inizializzata** con nome esterno non specificato.
Nella seconda dichiarazione `nameLst` è una lista **non-inizializzata** con nome esterno `NameLstExt`.
Nella terza dichiarazione `nameLst` è una lista con nome non specificato e con elementi `arr`.
Nella quarta dichiarazione `nameLst` è una lista con nome esterno `NameLstExt` e con elementi `arr`.
Nella quinta dichiarazione `nameLst` è l'insieme vuoto.

Vediamo alcuni esempi più specifici sulla definizione di liste logiche:

```
int[] elements = {1,1,2,3};
Lst L = new Lst("L", elements);
```

```
Lst R = new Lst ();
Lst Y = R.insAll(elements);
```

Nella prima dichiarazione si istanzia un array di elementi interi chiamato `elements`. Nella seconda si definisce la lista $L = [1, 2, 3, 3]$ di nome `L`. Nella terza si definisce la lista **non-bounded** $Y = [1, 2, 3, 3|R]$, con elementi specificati `elements` ed il resto degli elementi non specificati.

1.4 La nozione di multi-insieme

Definizione 1.5 *Un* ***multi-insieme logico*** *in JSetL è un'istanza della classe* `MultiSet`*, creata dalla dichiarazione*

```
MultiSet nameMSet = new MultiSet(nameMSetExt, ValueMSetElem);
```

dove `nameMSet` è il nome dell'oggetto Java, `nameMSetExt` è un nome esterno (parametro opzionale), `ValueMSetElem` è usato per specificare gli elementi del multi-insieme (parametro opzionale) e può essere un array di elementi. Se `ValueMSetElem` non è specificato viene costruito un multi-insieme non inizializzato. Il multi-insieme vuoto è denotato dalla costante `MultiSet.empty`. Le dichiarazioni hanno quindi la forma (con `arr` array e `name` una stringa)

```
MultiSet MSet = new MultiSet();
MultiSet MSet = new MultiSet(name);
MultiSet MSet = new MultiSet(arr);
MultiSet MSet = new MultiSet(name, arr);
MultiSet MSet = MultiSet.empty;
```

Nella prima dichiarazione `MSet` è un insieme **non-inizializzato** e con nome esterno non specificato.
Nella seconda dichiarazione `MSet` è un insieme **non-inizializzato** e con nome esterno `name`.
Nella terza dichiarazione `MSet` è un insieme con nome esterno non specificato e con elementi `arr`.

Nella quarta dichiarazione `MSet` è un insieme con nome esterno `name` e con elementi `arr`.
Nella quinta dichiarazione `MSet` è il `multi-insieme vuoto`.

1.5 Insiemi, Liste e Multi-insiemi in JSetL 1.3.2

Definizione 1.6 *Un insieme (o lista o multi-insieme) che contiene alcuni elementi che non sono inizializzati è detto* ***insieme (o lista o multi-insieme) parzialmente specificato****.*

Definizione 1.7 *Gli elementi di un insieme (o lista o multi-insieme) che sono essi stessi insiemi (o liste o multi-insiemi) sono detti* ***insiemi (o liste o multi-insiemi) annidati****.*

Definizione 1.8 *Un insieme logico (o lista o multi-insieme) che ha tutti gli elementi specificati è detto* ***insieme logico (o lista o multi-insieme) ground****.*

Variabili logiche, insiemi logici, liste logiche e multi-insiemi logici sono per definizione immutabili.

Insiemi, liste e multi-insiemi logici possono essere costruiti dinamicamente attraverso appositi metodi di inserimento ed estrazione che non modificano l'oggetto di invocazione ma ne restituiscono uno nuovo.

Definizione 1.9 *Un'espressione che costruisce e restituisce un nuovo insieme (o lista o multi-insieme) è detta* ***costruttore di insiemi (o liste o multi-insiemi)****.*

Siano `MSet` un multi-insieme, `n` un intero e `intArr` un array di interi. Sono ad esempio costruttori le seguenti espressioni:

```
MSet.ins(n);
```

```
MSet.insAll(intArr);
```

La prima espressione costruisce un multi-insieme uguale ad `MSet` con inserito l'elemento `n`. La seconda costruisce un multi-insieme uguale ad MSet con inseriti gli elementi dell'array `intArr`.

Gli insiemi, i multi-insiemi e le liste differiscono per queste proprietà:

- Negli insiemi non conta l'ordine degli elementi e non conta il numero di occorrenze degli elementi.
- Nei multi-insiemi non conta l'ordine, ma contano le occorrenze.
- Nelle liste conta sia l'ordine che le occorrenze.

Quindi le definizioni (e le relative implementazioni) delle operazioni su questi tipi di oggetti differiranno, una dall'altra, in modo tale da garantire le diverse proprietà.

JSetL prevede anche di gestire la risoluzione dei **vincoli**, tipica dei linguaggi CLP, su insiemi, multi-insiemi e liste. Si possono gestire vincoli di uguaglianza, disuguaglianza, appartenenza, unione, disgiunzione e altri. Siano `x` una variabile logica, S1, S2 ed S3 degli insiemi, `M` un multi-insieme. Le espressioni

```
x.in(S1);
x.in(M);
x.eq(S1)
S1.union(S2, S3);
```

sono vincoli atomici in JSetL.
La prima espressione costruisce il vincolo $x \in S_1$. La seconda costruisce il vincolo $x \in M$. La terza costruisce il vincolo $x = S_1$. La quarta costruisce il vincolo $S_1 = S_2 \cup S_3$.

Un'altra caratteristica della programmazione a vincoli è quella di prevedere un Solver (risolutore di vincoli), che si occupa di gestire la risoluzione dei vincoli. Siano `x` e `y` due variabili logiche, `S` un insieme e `Solver` il risolutore di vincoli

```
Solver.add(x.in(S));
Solver.add(y.in(S));
Solver.add(x.neq(y));
Solver.solve();
```

La prima espressione costruisce il vincolo $x \in S$ e lo aggiunge al Constraint Store (l'insieme dei vincoli). La seconda costruisce $y \in S$ e lo aggiunge al Constraint Store. La terza costruisce $x \neq y$ e lo aggiunge al Constraint Store. La quarta avvia il processo di risoluzione dei vincoli presenti nel Constraint Store.

1.6 Limiti e problemi di JSetL 1.3.2

In JSetL le variabili logiche hanno molte caratteristiche in comune con gli insiemi, i multi-insiemi e le liste. Ad esempio l'utilizzo dei vincoli di uguaglianza, disuguaglianza, appartenenza e non-appartenenza. Nonostante questo, nella versione attuale di JSetL, la definizione di variabile logica è "scollegata" dalle definizioni delle altre tre strutture. Anche a livello implementativo, non vi sono interfacce che possano offrire all'utente o al package una gestione uniforme.
Un altro aspetto da considerare è quello relativo alle caratteristiche comuni tra insiemi, multi-insiemi e liste. Queste caratteristiche sono date dalle proprietà e definizioni (1.6), (1.7), (1.8) e (1.9) oppure date dai vincoli di: unione, disgiunzione, intersezione, differenza. Anche in questo caso, nella versione di JSetL 1.3.2, non vi sono interfacce che possano permettere di utilizzare queste proprietà in modo uniforme.

Uno degli obiettivi della tesi è quello di permettere il trattamento uniforme di insiemi, liste e multi-insiemi. È necessario quindi introdurre, nella nuova architettura di JSetL, nuove definizioni e nuovi concetti, come ad esempio le interfacce.

Capitolo 2

Una nuova architettura per integrare insiemi, multi-insiemi e liste

Un obbiettivo di questa tesi è quello di costruire una o più interfacce che permettano, a livello utente (interfacce pubbliche) o internamente al package (interfacce protected), di unire i concetti comuni fra variabili logiche, insiemi, liste e multi-insiemi. È necessario quindi evidenziare le caratteristiche e le funzionalità comuni a queste quattro strutture. Concettualmente gli insiemi, le liste e i multi-insiemi possono essere considerati variabili logiche. È quindi necessaria la creazione dell'interfaccia `LVar` e l'uso dell'ereditarietà per la definizione delle classi di insiemi, liste e multi-insiemi. Inoltre gli insiemi, le liste e i multi-insiemi hanno caratteristiche comuni, riconducibili alla nozione generale di collezione logica. Sarà definita quindi la **collezione logica** e verrà creata l'interfaccia `LCollection`, da cui saranno derivate tramite ereditarietà le classi per insiemi, liste e multi-insiemi.

2.1 Variabili logiche

La nuova architettura di JSetL 1.3.3 prevede l'interfaccia `LVar`, che unisce i concetti comuni tra insiemi, liste, multi-insiemi e variabili logiche.

Le proprietà che caratterizzano una `variabile logica` sono:

- *Avere un nome*:
  Metodi: `setName`, `getName`.
- *Essere inizializzata o non-inizializzata*:
  Metodi: `setInit`, `isKnown`.
- *Essere immutabile*:
  Garantito dall'implementazione dei metodi.
- *Avere un valore*:
  Metodi: `getValue, setValue`.
- *Essere ground oppure no*:
  Metodi: `isGround`.
- *Poter definire un concetto di ugualianza tra due variabili logiche*:
  Metodi: `equals`.
- *Avere una funzione che trasferisca la rappresentazione della variabile logica in un canale di output:*
  Metodi: `output`.
- *Avere una rappresentazione sotto forma di stringa*:
  Metodi: `toString`.
- *Avere la gestione dei vincoli di ugualianza, disugualianza, appartenenza e non appartenenza:*
  Metodi: `eq, neq, in, nin`.

L'interfaccia `LVar` avrà perciò questi metodi per rispettare le proprietà elencate. Per maggiori dettagli si veda l'appendice A.1.

La definizione di variabile logica in JSetL diventa pertanto:

Definizione 2.1 *Una* ***variabile logica*** *è un'instanza di una classe che implementa l'interfaccia* `LVar`*, creata dalla dichiarazione*

```
LVar nameVar = new Class(NameVarExt, ValueVar);
```

dove `nameVar` è il nome dell'oggetto Java, `NameVarExt` (parametro opzionale) un nome esterno, `ValueVar` (parametro opzionale) è un valore di inizializzazione della variabile stessa. `Class` è una delle classi che implementano l'interfaccia LVar. In particolare è stata creata l'implementazione `ConcreteLVar`, che garantisce la compatibilità con la vecchia gestione delle variabili logiche in JSetL 1.3.2.

Ad esempio, siano `ConreteLVar` e `ConcreteLSet` due classi che implementano l'interfaccia `LVar` e `Solver` il risolutore di vincoli

```
int[] arr = {1,2,3};
LVar x = ConcreteLVar('x');
LVar S = ConcreteLSet('S', arr);
Solver.add(x.eq(S)); Solver.solve(); x.output(); S.output();
```

In questo esempio una variabile logica ed un insieme vengono trattati uniformemente tramite l'interfaccia `LVar`.

2.2 Collezioni logiche

Nella vecchia architettura di JSetL, il concetto di **collezione logica** non è presente. Si vuole invece realizzare un'interfaccia, chiamata `LCollection`, che permetta di unire i concetti comuni agli insiemi, liste e multi-insiemi, quindi a tutte le collezioni che abbiano queste caratteristiche.
Inoltre la **collezione logica** può vedersi come un caso particolare di variabile logica e quindi l'interfaccia `LCollection`, che offre più servizi all'utente,

estende l'interfaccia `LVar`. Le proprietà che caratterizzano la **collezione logica** sono:

- *Le proprietà che caratterizzano la* ***variabile logica***:
  `LCollection` estende `LVar`.
- *Essere una collezione vuota*:
  Metodi: `getEmpty`, `isEmpty`.
- *Avere un vettore di elementi*:
  Metodi: `getList`, `setList`.
- *Avere un puntatore ad una collezione logica che può rappresentare la parte non specificata della collezione*:
  Metodi: `getRest`, `setRest`.
- *Avere una rappresentazione come vettore*:
  Metodi: `toVector`, `get`, `getFirst`, `sub`, `subFirst`.
- *Essere limitato*:
  Metodi: `isBound`.
- *Avere un certo numero di elementi*:
  Metodi: `size`.
- *Essere ottenuto da un'altra collezione, con l'aggiunta di uno o più elementi*:
  Metodi: `ins`, `insAll`.
- *Essere trattabili tramite vincoli insiemistici*:
  Metodi: `union, disj`.

Diamo quindi la definizione di **collezione logica** in JSetL:

Definizione 2.2 *Una* ***collezione logica*** *è un'instanza di una classe che implementa l'interfaccia* `LCollection`*, creata dalla dichiarazione*

```
LCollection name = new Class(nameExt, Value);
```

dove `Class` è una delle classi che implementano l'interfaccia `LCollection`, `name` è il nome dell'oggetto Java, `nameExt` (parametro opzionale) un nome esterno, `Value` (parametro opzionale) è usato per specificare gli elementi della collezione e può essere un array. Se `Value` non è specificato viene costruita una collezione non-inizializzata. Le dichiarazioni hanno quindi la forma (con `arr` array di un qualsiasi tipo e `Class` una classe che implementa `LCollection`)

```
LCollection name = new Class();
LCollection name = new Class(nameExt, arr);
LCollection name2 = name.insAll(arr);
```

La prima dichiarazione costruisce una collezione non-inizializzata e con nome esterno non definito. La seconda costruisce una collezione con nome esterno `nameExt` ed elementi `arr`. La terza costruisce una collezione senza nome esterno definito, con elementi `arr` e con resto la collezione di nome `name`, questa ultima collezione sarà `non-bounded`.
L'interfaccia `LCollection` è descritta in appendice A.3.

2.3 La nuova architettura di JSetL

Oltre alle due interfacce `LVar` ed `LCollection` nella nuova architettura sono presenti anche le interfacce `LSet` (A.5), `LLst` (A.9) e `LBag` (A.7), che estendono `LCollection` e permettono di gestire uniformemente gli insiemi (`LSet`), le liste (`LLst`), i multi-insiemi (`LBag`). Per ognuna delle tre interfacce è presente anche una classe di implementazione, denominata rispettivamente `ConcreteLSet`, `ConcreteLLst` e `ConcreteLBag`.

Si può rappresentare la nuova architettura come nella figura 2.1.

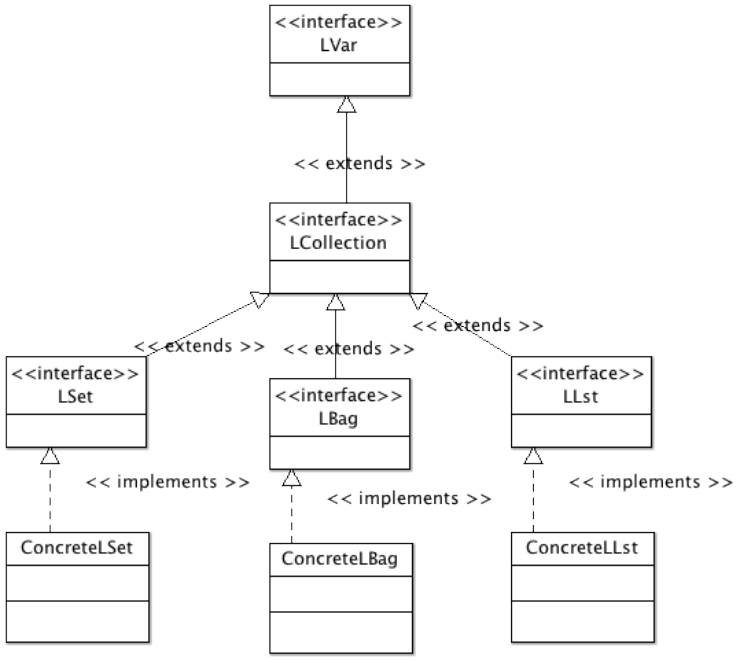


Figura 2.1: Architettura

Vediamo alcuni esempi di utilizzo delle interfacce:

```
int[] elements = {1,2,3};
LBag B = new ConcreteLBag("B", elements);
LLst L = new ConcreteLLst("L", elements);
LSet S = new ConcreteLSet("S", elements);
LSet MS = new ConcreteMutableLSet("MS", elements);
```

Nella prima dichiarazione si istanzia un array di elementi interi chiamato `elements`.

Nella seconda si definisce il multi-insieme logico $B = \{\!\{1, 2, 3\}\!\}$. Nella terza si definisce la lista logica $L = [1, 2, 3]$. Nella quarta e nella quinta si definiscono rispettivamente gli insiemi logici $S = \{1, 2, 3\}$ e $MS = \{1, 2, 3\}$. Nella seconda dichiarazione l'oggetto `B` è di tipo `LBag`, perciò da `B` si potranno invocare solo i metodi, lato utente, definiti nell'interfaccia `LBag` ed implementati in `ConcreteLBag`.

Analogo comportamento si ha per gli oggetti `L` ed `S`, interfacciati rispettivamente da `LLst` ed `LSet`. Nella quarta dichiarazione i metodi implementati in `ConcreteMutableLSet` sono interfacciati dalla classe `LSet`. Si noti che gli oggetti `S` ed `MS` saranno utilizzati allo stesso modo da parte dell'utente, anche se le implementazioni dell'interfaccia sono completamente diverse.

2.3.1 Separazione tra interfacce utente ed interfacce interne

Una delle caratteristiche di un linguaggio orientato agli oggetti, e quindi anche di Java, è di distinguere la visibilità dei metodi, all'interno e all'esterno del package. Nel caso di JSetL, in particolare, ci sono dei metodi offerti dal package che sono utilizzati all'interno del package stesso, ma che non si vuole siano visibili al generico utente della libreria. Per questo motivo sono state implementate altre cinque interfacce, `LVarProtected`, `LCollectionProtected`, `LSetProtected`, `LLstProtected` e `LBagPotected`. In esse sono definiti tutti quei metodi di utilità che possono essere usati esclusivamente all'interno del package. Viceversa, tutti i metodi di supporto alla programmazione dichiarativa oppure quei metodi che hanno utilità base, come possono essere i metodi di input e output, che devono essere visibili all'utente, sono definiti esclusivamente all'interno delle cinque interfacce generali, introdotte sopra: `LVar`, `LCollection`, `LSet`, `LLst` e `LBag`. Le interfacce protected possono essere utilizzate all'interno del package, per gestire in modo uniforme le strutture e facilitare la gestione dei dettagli implementativi interni alla libreria. Si può rappresentare la nuova architettura, comprendente anche le classi protected, come nella figura 2.2.
Quindi si ha che:

- `LVarProtected` estende `LVar`
- `LCollectionProtected` estende `LCollection` e `LVarProtected`
- `LSetProtected` estende `LSet` e `LCollectionProtected`
- `LBagProtected` estende `LBag` e `LCollectionProtected`

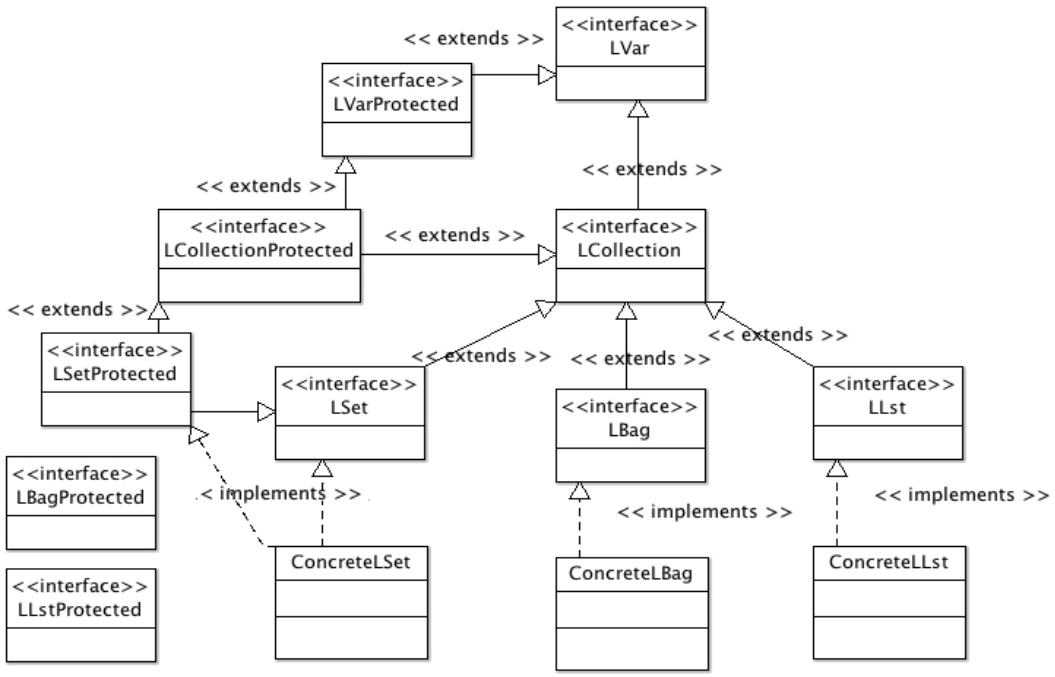


Figura 2.2: Architettura con classi protected

- `LLstProtected` estende `LLst` e `LCollectionProtected`

Infine `ConcreteLSet` implementa `LSetProtected`, `ConcreteLBag` implementa `LBagProtected`, `ConcreteLLst` implementa `LLstProtected`.
Vediamo alcuni esempi di utilizzo delle interfacce:

```
\\ esempio di metodo visibile dall'utente
foo(LVar x, LVar y, LCollection c){
    ...
    LCollection c2 = c.ins(x).ins(y); // c2 = {x,y|c};
    ...
}

\\ esempio di metodo visibile solo all'interno del package
bar(LVarProtected x, LCollectionProtected c){
    ...
```

```
    LVarProtected t1 = x.getValue();
    LVarProtected t2 = c.subFirst();
    foo(t1, t2, c);
    ...
}
```

La funzione `foo` rappresenta una funzione visibile dall'utente, in cui vengono usate le interfacce `LVar` ed `LCollection`. All'interno di questa funzione possono essere utilizzati i metodi delle due interfacce. La funzione `bar` rappresenta una funzione visibile solo all'interno del package. Per questo sono utilizzate le interfacce protected `LVarProtected` ed `LCollectionProtected`. Nel corpo della funzione possono essere invocati i metodi delle due interfacce e può essere chiamata la funzione `foo` con parametri attuali che usano interfacce protected grazie al meccanismo di ereditarietà ed estensione delle interfacce.

2.4 L'utilizzo delle nuove interfacce

Vediamo ora alcuni esempi di utilizzo delle interfacce presenti nella nuova architettura di JSetL all'interno del package JSetL stesso. Nella libreria sono già presenti vari casi in cui si possono utilizzare le nuove interfacce.

Un esempio è costituito dalla funzione di riscrittura del vincolo di disgiunzione. Questa funzione ha la stessa logica di funzionamento sia per gli insiemi che per i multi-insiemi. Senza l'interfaccia `LCollectionProtected` sarebbero necessarie due interfacce diverse per differenziare i due casi: la prima utilizzerebbe la vecchia classe `MultiSet`, la seconda `LSet`. Con l'utilizzo dell'interfaccia `LCollectionProtected`, invece, è possibile implementare un'unica funzione per la riscrittura del vincolo di disgiunzione: quando questa funzione verrà chiamata, come parametri attuali verrano passati, nel caso degli insiemi, due `ConcreteLSet`, nel caso dei multi-insiemi, due `ConcreteLBag`. All'interno della funzione verranno utilizzati i metodi messi a disposizione dall'interfaccia `LCollectionProtected`. A seconda della

classe che implementa `LCollectionProtected`, i metodi a loro volta avranno una implementazione diversa.
Possiamo rappresentare la funzione in questo modo:

```
protected void disj(LCollectionProtected collection1,
                LCollectionProtected collection2){
    ...
    LCollection c = collezione1.ins(2); // operazione *)
    ...
    // utilizzo interfaccia LCollectionProtected
    ...
}
```

Vediamo un esempio di utilizzo, all'interno del package, di questa funzione tramite le interfacce protected:

```
LCollectionProtected x1 = new ConcreteLSet();
LCollectionProtected x2 = new ConcreteLSet();

LCollectionProtected y1 = new ConcreteLBag();
LCollectionProtected y2 = new ConcreteLBag();

disj(x1, x2); // chiamata 1)
disj(y1, y2);// chiamata 2)
```

Se si considera la chiamata 1), quando viene eseguita l'operazione `*)` a tempo di esecuzione viene chiamato il metodo `ins` della classe `ConcreteLSet`. Nel caso della chiamata 2), viene eseguito a tempo di esecuzione il metodo `ins` della classe `ConcreteLBag`.
L'implementazione dettagliata del metodo `disj` è nella Sezione 4.5.

Capitolo 3

Definizione e trattamento dei vincoli sui multi-insiemi

In questo Capitolo viene dapprima introdotta, in modo formale, la nozione di multi-insieme e definite le relative operazioni e proprietà generali. Vengono quindi introdotti i vincoli di base sui multi-insiemi, appartenenza multipla, unione somma ($\uplus$) e disgiunzione ($||$), e definite le relative procedure di risoluzione dei vincoli, in modo analogo a quanto fatto per gli insiemi in [2] ed estendendo i risultati sui multi-insiemi presentati in [3]. Successivamente viene mostrato (e dimostrato formalmente) che i vincoli di $\subseteq, \cap, \cup, \setminus$ su multi-insiemi possono essere espressi tramite congiunzioni di vincoli basati soltanto su $\uplus$ e $||$, e quindi facilmente implementati in termini di quest'ultimi.

3.1 Introduzione ai multi-insiemi

Un **multi-insieme** (o **bag**) è una collezione non ordinata di oggetti (o elementi) in cui, diversamente dagli insiemi Cantoriani, gli elementi possono ripetersi. In altre parole in un multi-insieme, gli elementi possono ripetersi più di una volta e qundi esso non ha le stesse proprietà di un insieme Cantoriano. Il numero di volte che un elemento si ripete in un multi-insieme

viene chiamato **molteplicità**. Il numero di elementi distinti presenti in un multi-insieme e le loro molteplicità determinano, insieme, la sua cardinalità. Un multi-insieme è chiamato finito se sono finiti il numero degli elementi distinti e la molteplicita di ognuno di essi, altrimenti si dice infinito.

Un multi-insieme può essere visto come una funzione $\alpha : P \to O$, dove concettualmente P è l'insieme delle presenze e O è l'insieme numerico delle occorrenze.
α rappresenta:

$$\begin{cases} \text{un insieme se } O = \{0, 1\}. \\ \text{un multi-insieme se } O = \mathbb{N}. \\ \text{un multi-insieme con segno se } O = \mathbb{Z}. \\ \text{un insieme-fuzzy se } O = [0, 1] \subseteq \mathbb{R}. \end{cases}$$

Per maggiori dettagli si veda l'articolo [4]. In questa tesi verrà trattato il caso $O = \mathbb{N}$.

3.2 Definizione di multi-insieme

Definizione 3.1 *(Multi-insieme)*

*Siano P l'insieme delle presenze e $O \subseteq \mathbb{N}$ l'insieme delle occorrenze. Un **multi-insieme** M è una relazione funzionale tale che $M \subseteq P \times O$.*

Definizione 3.2 *(Molteplicità)*

*Si definisce **molteplicità** relativa ad M la funzione:*

$m_M : P \to O$

dove:

$$m_M(a) = b \quad \forall a \in P, \text{ con } \langle a, b \rangle \in M.$$

Talvolta è necessario estendere il dominio della funzione ponendo in tal caso:

$$m_M(c) = 0 \quad \forall c \notin P$$

Osservazione: Poichè M è funzionale allora m_M è una funzione ben definita di cui M è il grafico.

Definizione 3.3 *(Appartenenza multipla)*

Siano M un multi-insieme e $n \in \mathbb{N}$.
Si dice che x ***appartiene n volte*** *ad M e si scrive $x \in^n M$, se $m_M(x) = n$.*

3.2.1 Operazioni e proprietà

Definizione 3.4 *Siano A e B due multi-insiemi. Si dice che A è* ***uguale*** *a B e si scrive $A = B$ se:*

$$\forall c,\ m_A(c) = m_B(c). \tag{3.1}$$

Definizione 3.5 *Siano A e B due multi-insiemi. Si dice che A è* ***sottoinsieme*** *di B e si scrive $A \subseteq B$ se:*

$$\forall c,\ m_A(c) \leqslant m_B(c). \tag{3.2}$$

Definizione 3.6 *Siano A e B due multi-insiemi. Si chiama* ***unione somma*** *tra A e B, e si scrive $A \uplus B$ se:*

$$\forall c,\ m_{A \uplus B}(c) = m_A(c) + m_B(c). \tag{3.3}$$

Definizione 3.7 *Siano A e B due multi-insiemi. Si dice che A è* ***disgiunto*** *da B e si scrive $A||B$ se:*

$$\forall c,\ (m_A(c) \geqslant 1 \Longrightarrow m_B(c) = 0)\ \wedge\ (m_B(c) \geqslant 1 \Longrightarrow m_A(c) = 0). \tag{3.4}$$

Definizione 3.8 *Siano A e B due multi-insiemi. Si chiama* ***intersezione*** *tra A e B, e si scrive $A \cap B$ se:*

$$\forall c,\ m_{A \cap B}(c) = min(m_A(c), m_B(c)). \tag{3.5}$$

Definizione 3.9 *Siano A e B due multi-insiemi. Si chiama* ***unione massimo*** *tra A e B, e si scrive $A \cup B$ se:*

$$\forall c,\ m_{A \cup B}(c) = max(m_A(c), m_B(c)). \tag{3.6}$$

Definizione 3.10 *Siano A e B due multi-insiemi. Si chiama* ***differenza*** *tra A e B, e si scrive $A \backslash B$ se:*

$$\forall c,\ m_{A \backslash B}(c) = m_A(c) - m_{A \cap B}(c). \tag{3.7}$$

3.3 La riscrittura dei vincoli sui multi-insiemi

Uno degli obbiettivi di questa tesi è quello di estendere le funzionalità di JSetL relative alla gestione dei vincoli sui multi-insiemi. Si vogliono quindi trovare le regole di riscrittura che l'algoritmo deve applicare per la risoluzione dei vincoli di **appartenenza multipla**, **unione somma**, **disgiunzione**, **intersezione**, **sottoinsieme** e **differenza**.

Definizione 3.11 *(Vincolo atomico) In JSetL un* ***vincolo atomico*** *sui multi-insiemi è una relazione binaria o ternaria, definita mediante le espressioni:*

- $\mathsf{op}(e_1, e_2)$;
- $\mathsf{op}(e_1, e_2, e_3)$.

dove op è uno dei seguenti operatori eq, neq, in, nin, union, inters, disj, differ, unionMax, *ed e_1, e_2, e_3 sono espressioni il cui tipo dipende da* op.

Definizione 3.12 *(Vincolo) Un* ***vincolo*** *(constraint) è una congiunzione di due o più vincoli atomici.*

I vincoli possono essere aggiunti ad un Constraint Store (insieme di vincoli), che è considerato una congiunzione di vincoli. Dato un Constraint Store, esiste un algoritmo di risoluzione che riscrive i vincoli fino a trovare la forma risolta. Ad ogni passo di questo algoritmo viene applicata una

regola di riscrittura del vincolo presente nel Constraint Store. Ogni vincolo ha le proprie regole di riscrittura e sono queste regole a determinare le caratteristiche del vincolo. Le regole di riscrittura utilizzate nell'algoritmo di risoluzione hanno la seguente forma generale.

$$\frac{\textit{pre-condizioni}}{\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}}$$

dove $C_1, \ldots, C_n$ e $C'_1, \ldots, C'_m$ ($n,\ m \geq 0$) sono vincoli, e $\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}$ rappresenta il cambiamento nel **constraint store** causato dall'applicazione della regola.
In seguito verranno adottate le seguenti convenzioni:

- t, t_i, s, s_i, u, u_i, m, m_i, n: qualsiasi termine (ground o non-ground).
- k: costante intera.
- T, S, U, M, N: variabili (non-inizializzate).

Inoltre assumiamo che un termine *ground* rappresenti un oggetto (ad esempio un multi-insieme) completamente specificato (o semplicemente **noto**); un termine *non ground* (ovvero contenente variabili non inizializzate) rappresenti un oggetto (ad esempio un multi-insieme) **parzialmente specificato**; una variabile non inizializzata rappresenti un oggetto **non noto**.

Di seguito sono presentate le tabelle che descrivono le regole di riscrittura dei vincoli in, unione disjin cui assumiamo che:

- $\mathsf{in}(t, n, m)$ sta per $t \in^n m$ e $t \notin m$ sta per $t \in^0 m$;
- $\mathsf{union}(t, s, u)$ sta per $u = t \uplus s$;
- $\mathsf{disj}(t, s)$ sta per $t||s$, $t \in s$ per $t \in^n s$ con $n \geqslant 1$ ed $t \notin s$ sta per $t \in^0 s$.

3.3.1 Vincolo di appartenenza multipla

- *multi-insieme vuoto*

$$\frac{}{\{\mathsf{in}(t, \emptyset, n)\} \rightarrow \{n = 0\}} \tag{3.8}$$

- *multi-insieme vuoto*

$$\frac{m \neq \emptyset}{\{\mathsf{in}(t, m, 0)\} \rightarrow \{t \notin m\}} \tag{3.9}$$

- *multi-insieme noto*

$$\frac{t \text{ noto } \wedge \ m \text{ noto } \wedge \ k = count(t, m)}{\{\mathsf{in}(t, m, n)\} \rightarrow \{n = k\}} \tag{3.10}$$

  dove `count(t,m)` è una funzione che conta il numero di occorrenze di t in m.

- *caso deterministico*

$$\frac{}{\{\mathsf{in}(t, M, k)\} \rightarrow \{M = \{\!\!\{ t \underbrace{\cdots}_{k} t | R \}\!\!\}, \ t \notin R\}} \tag{3.11}$$

- *caso non deterministico*

$$\frac{}{\begin{array}{c}\{\mathsf{in}(t, \{\!\!\{ m_1 | m_2 \}\!\!\}, n)\} \rightarrow \{t = m_1, \ \mathsf{in}(t, m_2, N), \ N = n - 1\} \ or \\ \{\mathsf{in}(t, \{\!\!\{ m_1 | m_2 \}\!\!\}, n)\} \rightarrow \{t \neq m_1, \ \mathsf{in}(t, m_2, n)\}\end{array}} \tag{3.12}$$

3.3.2 Vincolo di unione somma

- *multi-insieme vuoto*

$$\frac{}{\{\mathsf{union}(t, s, \emptyset)\} \rightarrow \{t = \emptyset, \ s = \emptyset\}} \tag{3.13}$$

- *multi-insieme vuoto*

$$\frac{u \neq \emptyset}{\{\mathsf{union}(t, \emptyset, u)\} \rightarrow \{t = u\}} \tag{3.14}$$

- *multi-insieme vuoto*

$$\frac{u \neq \emptyset \wedge s \neq \emptyset}{\{\mathsf{union}(\emptyset, s, u)\} \rightarrow \{s = u\}} \tag{3.15}$$

- *caso deterministico*

$$\frac{}{\{\mathsf{union}(T, \{\!|\, s_1|s_2 \,|\!\}, u)\} \rightarrow \{u = \{\!|\, s_1|R \,|\!\},\ \mathsf{union}(T, s_2, R)\}} \tag{3.16}$$

- *caso deterministico*

$$\frac{}{\{\mathsf{union}(\{\!|\, t_1|t_2 \,|\!\}, s, u)\} \rightarrow \{u = \{\!|\, t_1|R \,|\!\},\ \mathsf{union}(t_2, s, R)\}} \tag{3.17}$$

- *caso non deterministico*

$$\frac{}{\begin{array}{c}\{\mathsf{union}(T, S, \{\!|\, u_1|u_2 \,|\!\})\} \rightarrow \{T = \{\!|\, u_1|R \,|\!\}, \mathsf{union}(R, S, u_2)\}\ \mathit{or} \\ \{\mathsf{union}(T, S, \{\!|\, u_1|u_2 \,|\!\})\} \rightarrow \{S = \{\!|\, u_1|R \,|\!\}, \mathsf{union}(T, R, u_2)\}\end{array}} \tag{3.18}$$

3.3.3 Vincolo di disgiunzione

- *multi-insieme vuoto*

$$\frac{}{\{\mathsf{disj}(\emptyset, s)\} \rightarrow \mathit{true}} \tag{3.19}$$

- *multi-insieme vuoto*

$$\frac{t \neq \emptyset}{\{\mathsf{disj}(t, \emptyset)\} \rightarrow \mathit{true}} \tag{3.20}$$

- *multi-insieme uguale*

$$\frac{t \neq \emptyset}{\{\mathsf{disj}(t, t)\} \rightarrow \{t = \emptyset\}} \tag{3.21}$$

- *parzialmente definito*

$$\frac{}{\{\mathsf{disj}(\{\!|\, t_1|t_2 \,|\!\}, S)\} \rightarrow \{t_1 \notin S,\ t_2||S\}} \tag{3.22}$$

- *parzialmente definito*

$$\frac{}{\{\mathsf{disj}(T, \{\!|\, s_1|s_2 \,|\!\})\} \rightarrow \{s_1 \notin T,\ T||s_2\}} \tag{3.23}$$

- *parzialmente definito*

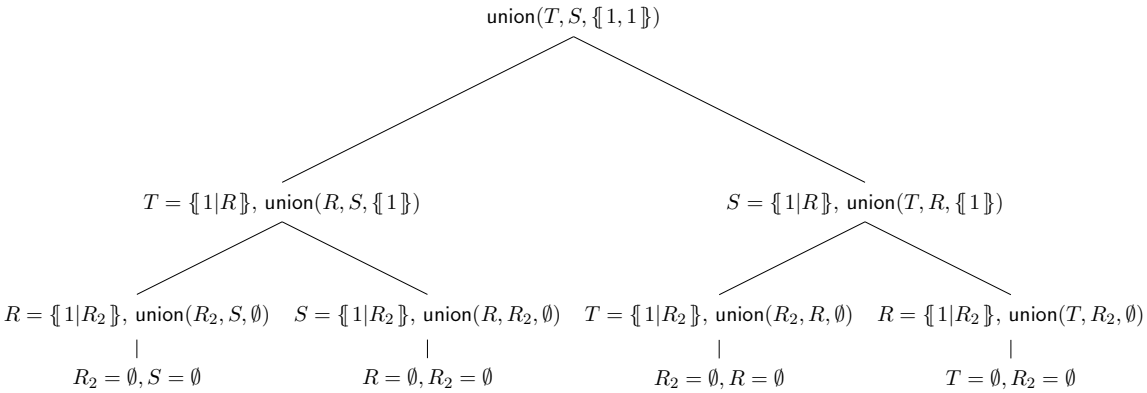
$$\frac{}{\{\mathsf{disj}(\{\!|\, t_1|t_2 \,|\!\}, \{\!|\, s_1|s_2 \,|\!\})\} \rightarrow \{t_1 \neq s_1,\ t_1 \notin s_2,\ s_1 \in t_2,\ t_2||s_2\}} \tag{3.24}$$

3.4 Un esempio

Si vuole ora fare un esempio del funzionamento dell'algoritmo di risoluzione e di applicazione delle regole per il vincolo **unione somma**.
Siano T ed S due multi-insiemi, si calcolano le soluzione del vincolo

$$\mathsf{union}(T, S, \{\!\{1, 1\}\!\})$$

nel seguente modo



Ogni ramo dell'albero rappresenta l'applicazione di una delle regole della tabella 3.3.2. Ogni nodo rappresenta l'insieme dei vincoli in una data situazione. I cammini dalle foglie alla radice rappresentano le soluzioni, percorrendo un cammino possiamo costruire il valore delle variabili. In questo caso le soluzioni saranno:

$T = \{\!\{1, 1\}\!\},\ \ S = \emptyset$;

$T = \{\!\{1\}\!\},\ \ S = \{\!\{1\}\!\}$;

$T = \{\!\{1\}\!\},\ \ S = \{\!\{1\}\!\}$;

$T = \emptyset,\ \ S = \{\!\{1, 1\}\!\}$.

3.5 Definizione di altri vincoli sui multi-insiemi

Nelle Sezioni precedenti abbiamo introdotto i vincoli di **appartenenza multipla**, **unione somma** e **disgiunzione** e definito le relative regole di

riscrittura. In questo Capitolo si mostra che è possibile realizzare le altre operazioni sui multi-insiemi (precisamente **intersezione**, **sottoinsieme**, **unione massimo** e **differenza**) come vincoli definiti soltanto in termini dei vincoli atomici già introdotti. In questo modo le procedure di riscrittura di questi nuovi vincoli diventano banali. I risultati illustrati in questo Capitolo sono del tutto generali e si basano sulla definizione astratta delle operazioni su multi-insiemi data nella Sezione 3.2.

Si vuole dimostrare che:

Teorema 3.5.1 *Per ogni formula logica in cui compaiono le operazioni* $\subseteq, \cap, \cup, \setminus$*, esiste una formula logica tautologicamente equivalente in cui compaiono solo le operazioni* $\uplus$ *e* $||$*.*

Per dimostrare questo teorema si dimostreranno prima quattro lemmi:

Lemma 1 *Siano* r, s, t *tre multi-insiemi:*

$$t = r \cap s \Longleftrightarrow \exists R, S(r = R \uplus t \wedge s = S \uplus t \wedge R||S).$$

Dimostrazione. ($\Longleftarrow$) Dall'ipotesi e dalle definizioni si ricavano le seguenti proprietà: $\forall c$,

$$m_r(c) = m_R(c) + m_t(c) \tag{3.25}$$

$$m_s(c) = m_S(c) + m_t(c) \tag{3.26}$$

$$(m_R(c) \geqslant 1) \Longrightarrow (m_S(c) = 0) \wedge (m_S(c) \geqslant 1) \Longrightarrow (m_R(c) = 0). \tag{3.27}$$

- *Caso I:*

$$\forall c,\ (m_R(c) \geqslant 1) \xrightarrow{(3.27)} (m_S(c) = 0)$$

$$\xrightarrow{(3.26)e(3.25)} m_s(c) = m_t(c)\ \wedge\ m_t(c) \leqslant m_r(c)$$

$$\Longrightarrow m_t(c) = min(m_r(c), m_s(c))$$

$$\xrightarrow{(3.5)} t = r \cap s.$$

- *Caso II:*

$$\forall c,\ (m_S(c) \geqslant 1) \xRightarrow{(3.27)} (m_R(c) = 0).$$

  Questo caso è uguale al precedente scambiando r con s ed R con S.

- *Caso III:*

$$\forall c,\ (m_R(c) = 0) \wedge (m_S(c) = 0)$$
$$\xRightarrow{(3.26)e(3.25)} m_t(c) = m_r(c) = m_s(c)$$
$$\Longrightarrow m_t(c) = min(m_r(c), m_s(c))$$
$$\xRightarrow{(3.5)} t = r \cap s.$$

($\Longrightarrow$) Pongo:

$$R = r \backslash t$$
$$S = s \backslash t$$

da cui segue

$$\xRightarrow{(3.7)} \forall c, m_R(c) = m_r(c) - min(m_r(c), m_t(c)) \wedge m_S(c) = m_s(c) - min(m_s(c), m_t(c))$$
$$\xRightarrow{HP} m_R(c) = m_r(c) - m_t(c) \ \wedge \ m_S(c) = m_s(c) - m_t(c).$$

- *Caso I:*

$$\forall c, m_R(c) \geqslant 1 \Longrightarrow m_t(c) < m_r(c) \Longrightarrow m_t(c) = m_s(c) \Longrightarrow m_S(c) = 0.$$

- *Caso II:*

$$\forall c, m_S(c) \geqslant 1 \Longrightarrow m_t(c) < m_s(c) \Longrightarrow m_t(c) = m_r(c) \Longrightarrow m_R(c) = 0.$$

- *Caso III:*

$$\forall c,\ (m_R(c) = 0) \wedge (m_S(c) = 0).$$

$\xRightarrow{(3.4)}$ In tutti e tre i casi $\exists R, S : R||S$.

□

Lemma 2 *Siano r, s, t tre multi-insiemi:*

$$t = r \backslash s \Longleftrightarrow \exists Z (r = t \uplus Z \ \wedge \ Z = r \cap s).$$

Dimostrazione. ($\Longleftarrow$) Dall'ipotesi e dalle definizioni si ricavano le seguenti proprietà:

$$\forall c, \ m_r(c) = m_t(c) + m_Z(c) \tag{3.28}$$

$$\forall c, \ m_Z(c) = m_{r \cap s}(c). \tag{3.29}$$

Dalla proprietà (3.28) usando la (3.29) ricavo

$$m_r(c) = m_t(c) + m_{r \cap s}(c)$$

$$\Longrightarrow m_r(c) - m_{r \cap s}(c) = m_t(c)$$

$$\stackrel{(3.7)}{\Longrightarrow} t = r \backslash \ s.$$

($\Longrightarrow$) Pongo:

$$Z = r \cap s$$

da cui

$$\stackrel{(3.1)}{\Longrightarrow} \forall c, \ m_Z(c) = m_{r \cap s}(c)$$

$$\stackrel{HP \ e \ (3.7)}{\Longrightarrow} m_t(c) = m_r(c) - m_Z(c)$$

$$\Longrightarrow m_r(c) = m_t(c) + m_Z(c)$$

$$\stackrel{(3.3)}{\Longrightarrow} r = t \uplus Z.$$

□

Lemma 3 *Siano r, s due multi-insiemi:*

$$s \subseteq t \Longleftrightarrow \exists X (t = X \uplus s).$$

Dimostrazione. ($\Longleftarrow$) Dall'ipotesi e dalle definizioni si ricava la seguente proprietà:

$$\forall c,\ m_t(c) = m_X(c) + m_s(c)$$
$$\xrightarrow{(m_X(c)\geqslant 0)} m_s(c) \leqslant m_t(c)$$
$$\xrightarrow{(3.2)} s \subseteq t.$$

($\Longrightarrow$) Pongo:

$$X = t \backslash s$$
$$\xrightarrow{(3.7)} \forall c,\ m_X(c) = m_t(c) - m_{t\cap s}(c)$$
$$\xrightarrow{(3.5)} m_X(c) = m_t(c) - min(m_t(c), m_s(c))$$
$$\xrightarrow{HP} m_X(c) = m_t(c) - m_s(c)$$
$$\Longrightarrow m_t(c) = m_X(c) + m_s(c)$$
$$\xrightarrow{(3.3)} \exists X : t = X \uplus s.$$

□

Lemma 4 *Siano r, s, t tre multi-insiemi:*

$$t = r \cup s \Longleftrightarrow \exists W, X(W = r \cap s \ \wedge\ X = r \uplus s \ \wedge\ t = X \backslash W).$$

Dimostrazione. ($\Longleftarrow$) Dall'ipotesi e dalle definizioni ricavo le seguenti proprietà:

$$\forall c, m_t(c) = m_X(c) - m_{X\cap W}(c) \tag{3.30}$$
$$\forall c, m_X(c) = m_r(c) + m_s(c) \tag{3.31}$$
$$\forall c, m_W(c) = m_{r\cap s}(c). \tag{3.32}$$

Per la proprietà (3.30):

$$\forall c,\ m_t(c) = m_X(c) - m_{X\cap W}(c)$$

$$\xRightarrow{(3.5)} m_t(c) = m_X(c) - min(m_X(c), m_W(c))$$

$$\xRightarrow{(3.31)} m_t(c) = m_r(c) + m_s(c) - min(m_r(c) + m_s(c), m_W(c))$$

$$\xRightarrow{(3.32)} m_t(c) = m_r(c) + m_s(c) - min(m_r(c) + m_s(c), m_{r\cap s}(c))$$

$$\xRightarrow{(3.5)} m_t(c) = m_r(c) + m_s(c) - min(m_r(c) + m_s(c), min(m_r(c), m_s(c)))$$

$$\Longrightarrow m_t(c) = m_r(c) + m_s(c) - min(m_r(c), m_s(c))$$

$$\Longrightarrow m_t(c) = max(m_r(c), m_s(c))$$

$$\xRightarrow{(3.6)} t = r \cup s.$$

($\Longrightarrow$) Pongo:

$$W = r \cap s \tag{3.33}$$

$$X = r \uplus s. \tag{3.34}$$

Per ipotesi si ha: $\forall c,\ m_t(c) = max(m_r(c), m_s(c))$

$$\Longrightarrow m_t(c) = m_r(c) + m_s(c) - min(m_r(c), m_s(c))$$

$$\Longrightarrow m_t(c) = m_r(c) + m_s(c) - min(min(m_r(c), m_s(c)), m_r(c) + m_s(c))$$

$$\xRightarrow{(3.34)} m_t(c) = m_X(c) - min(min(m_r(c), m_s(c)), m_X(c))$$

$$\xRightarrow{(3.33)} m_t(c) = m_X(c) - min(m_W(c), m_X(c))$$

$$\xRightarrow{(3.5)} m_t(c) = m_X(c) - m_{W\cap X}(c)$$

$$\xRightarrow{(3.7)} \exists W, X : t = X \backslash W.$$

□

Dimostrazione. (Teorema 3.5.1): Si può concludere, grazie ai lemmi 1, 2, 3, 4, che le operazioni $\cap, \backslash, \subseteq, \cup$ si possono ricavare dalle sole due operazioni $||$, $\uplus$.

□

Grazie a questo risultato teorico si possono trarre varie conseguenze relative alla riscrittura dei vincoli. La conseguenza più importante è che si possono implementare solo le regole di riscrittura per i vincoli di **unione somma** e **disgiunzione**, per poi definire le regole per tutti gli altri vincoli come semplice riscrittura di ciascun nuovo vincolo nell'insieme dei vincoli $||$ e $\uplus$ equivalente, in accordo con i risultati teorici ottenuti nei lemmi 1, 2, 3, 4. Si noti che formule ottenute sono semplici congiunzioni, quantificate esistenzialmente, di vincoli primitivi e quindi, loro stesse, vincoli.

Di seguito sono rappresentate le tabelle che descrivono le regole di riscrittura dei vincoli unionMax, inters, differ e subset, in cui assumiamo che:

$$\mathsf{unionMax}(r,s,t)\ sta\ per\ t = r \cup s;$$

$$\mathsf{inters}(r,s,t)\ sta\ per\ t = r \cap s;$$

$$\mathsf{differ}(r,s,t)\ sta\ per\ t = r \backslash s;$$

$$\mathsf{subset}(s,t)\ sta\ per\ s \subseteq t.$$

Regole di riscrittura dei vincoli.

- *intersezione*

$$\frac{}{\{\mathsf{inters}(r,s,t)\} \rightarrow \{\mathsf{union}(R,t,r),\ \mathsf{union}(S,t,s),\ \mathsf{disj}(R,S)\}} \tag{3.35}$$

- *differenza*

$$\frac{}{\{\mathsf{differ}(r,s,t)\} \rightarrow \{\mathsf{union}(t,Z,r),\ \mathsf{inters}(r,s,Z)\}} \tag{3.36}$$

- *sottoinsieme*

$$\frac{}{\{\mathsf{subset}(s,t)\} \rightarrow \{\mathsf{union}(s,X,t)\}} \tag{3.37}$$

- *unione massimo*

$$\frac{}{\{\mathsf{unionMax}(r,s,t)\} \rightarrow \{\mathsf{inters}(r,s,W),\ \mathsf{union}(r,s,X),\ \mathsf{differ}(X,W,t)\}} \tag{3.38}$$

Capitolo 4

Realizzazione dei vincoli sui multi-insiemi in JSetL

In questo Capitolo viene mostrato come implementare in JSetL le regole di riscrittura dei vincoli, relative alle tabelle che sono state ricavate dalla teoria dei multi-insiemi, presentate nel Capitolo 3.

4.1 Vincoli sui multi-insiemi in JSetL 1.3.3

Uno degli obbiettivi di questa tesi è quello di estendere le funzionalità di JSetL relative alla gestione dei vincoli sui multi-insiemi. I vincoli sui multi-insiemi che si vogliono implementare sono quelli di **appartenenza multipla**, **unione somma**, **disgiunzione**, **intersezione**, **sottoinsieme**, **differenza** e **unione massimo**.

Di seguito è riportato l'utilizzo, lato utente, dei metodi relativi ai vincoli che si vogliono implementare.
Siano m_1, m_2, m_3 tre multi-insiemi, l una variabile logica ed n una variabile logica intera. I vincoli considerati hanno la seguente forma:

- **Appartenenza multipla** $l.\mathsf{in}(m_1, n)$;
- **Unione somma** $m_1.\mathsf{union}(m_2, m_3)$;
- **Disgiunzione** $m_1.\mathsf{disj}(m_2)$;

- **Intersezione** m_1.inters(m_2, m_3);
- **Sottoinsieme** m_1.subset(m_2);
- **Differenza** m_1.differ(m_2, m_3);
- **Unione massimo** m_1.unionMax(m_2, m_3).

Questi vincoli sono realizzati come metodi dela classe `LBag` che restituiscono un tipo `Constraint` e, come visto alla fine della Sezione (1.5), è possibile aggiungere questi vincoli ad un Constraint Store (insieme di vincoli) implementato dalla classe `Solver` (risolutore di vincoli) tramite l'invocazione del metodo

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

dove `S` è un'istanza della classe `Solver` e `C` è un'istanza della classe `Constraint` oppure `ConstraintConjunction`. Alla fine è possibile cercare le soluzioni dei vincoli invocando il metodo:

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

4.2 Implementazioni dei vincoli in JSetL

Il metodo `solve()` implementa l'algoritmo di risoluzione. Vengono considerati tutti i vincoli presenti nel Constraint Store e in modo non-deterministico vengono cercate tutte le soluzioni. Se non viene trovata nessuna soluzione viene lanciata l'eccezione `Failure`. Il metodo `solve()` utilizza la classe `RewritingConstraintsRules` per implementare l'algoritmo di risoluzione dei vincoli. Ad ogni passo viene controllato un vincolo nel Constraint Solver e, a seconda del tipo di vincolo, viene invocato un metodo in `RewritingConstraintRules` che implementa la riscrittura di quel vincolo. Di seguito vengono presentate le implementazioni di questi metodi per i vincoli sui multi-insiemi sopra elencati.

4.3 Vincolo di appartenenza multipla

Le regole di riscrittura del vincolo di **appartenenza multipla**, descritte nella tablella (3.3.1), sono implementate nel metodo `inNMultiset` della classe `RewritingConstraintsRules`, la testata del metodo ha il seguente aspetto:

```
private void inNMultiset(LVarProtected lvar, LBagProtected m,
                         LVarProtected N, Constraint s) throws Failure{
```

dove *lvar* appartiene N volte a m (`lvar.in(m,N);`).
Caso 3.8

```
if (m.isEmpty()) { // se il m e‘ vuoto => N = 0
    s.arg1 = N;
    s.cons = Enviroment.eqCode;
    s.arg2 = new ConcreteLVar(0);
    eq(s);
    Solver.storeInvariato = false;
    return;
 }
```

Se m è vuoto viene trattato il vincolo $N = 0$ con la chiamata
`eq(s)`.

Caso 3.9

```
if(N.equals(0)){   N = 0 => lvar.nin(m)
    s.arg1 = lvar;
    s.cons = Enviroment.ninCode;
    s.arg2 = m;
    nin(s);
    Solver.storeInvariato = false;
    return;
}
```

Se N è uguale a 0 viene trattato il vincolo $lvar \notin m$ con la chiamata
`nin(s)`.

Caso 3.10

```
else if(m.isGround() && lvar.isGround()){ // controlla se N = conta(lvar,m)
    Integer counter = 0;
    Vector vector_untail = m.untail().toVector();
    Iterator i = vector_untail.iterator();
    while(i.hasNext()) { // conta gli lvar in m
        if(((LVar)i.next()).equals(lvar)) counter++;
    }

    LVar Nnew = new ConcreteLVar(counter);
    s.arg1 = Nnew;
    s.cons = Enviroment.eqCode;
    s.arg2 = N;
    eq(s);
    Solver.storeInvariato = false;
    return;
}
```

Se m è ground e $lvar$ è definito allora viene costruita una `LVar` di nome `counter` che è uguale al numero di occorrenze di $lvar$ in m
`LVar Nnew = new ConcreteLVar(counter);`.

Viene trattato il vincolo $N = Nnew$ con la chiamata a funzione `eq(s)`.

Caso 3.11

```
else if(!m.isKnown() && N.isGround()){ //

    Integer Nint = (Integer) N.getValue();
```

```
    LVar[] arr = new LVar[Nint];

    for(Integer i = 0; i < Nint; i++) {
        arr[i] = lvar;
    }

    LBag R = new ConcreteLBag();
    s.arg1 = m;
    s.cons = Enviroment.eqCode;
    s.arg2 = R.insAll(arr);
    eq(s);
    Solver.storeInvariato = false;
    return;
}
```

Se N è definito e m è non-inizializzato allora viene costruito l'array `arr` contenente N volte la variabile $lvar$.

```
LVar[] arr = new LVar[Nint];

for(Integer i = 0; i < Nint; i++) {
    arr[i] = lvar;
}
```

Viene trattato il vincolo $m = \{\!\{arr|R\}\!\}$ con la chiamata a funzione `eq(s)`.

Caso 3.12

```
else if((!lvar.isGround() || !m.isGround()) && m.size() >= 1){ // m = {{t'|R}}
    switch (s.caseControl) {
    case 0: // (i)
        VarState statoVarIn = Solver.B.getVarState();
        StoreState statoStoreIn = Solver.B.getStoreState(s);
        Hashtable<ConcreteLVar, Interval>
```

```
                domainStateIn = Solver.B.getDomainState();

        Solver.B.addChoicePoint(1, statoVarIn, statoStoreIn, domainStateIn);

        Constraint s2 = new Constraint(lvar, Enviroment.eqCode,
                        m.getFirst()); //lvar = t'
        Solver.add(Solver.indexOf(s) + 1, s2);

        LVar M = new ConcreteLVar();
        Constraint s3 = new Constraint(Enviroment.subCode,
                        M, N, new ConcreteLVar(1)); // M = N-1
        Solver.add(Solver.indexOf(s2) + 1, s3);



        s.arg1 = lvar;
        s.arg2 = m.sub();
        s.arg3 = M;
        s.cons = Enviroment.inNCode;
        inN(s); // lvar.in(R, M)
        Solver.storeInvariato = false;
        return;
    case 1: // (ii)
        Constraint s5 = new Constraint(lvar, Enviroment.neqCode,
                        m.getFirst()); // lvar <> t'
        Solver.add(Solver.indexOf(s) + 1, s5);

        s.caseControl = 0;
        s.arg1 = lvar;
        s.arg2 = m.sub(); // R
        s.arg3 = N;
        s.cons = Enviroment.inNCode;
```

```
        inN(s); // lvar.in(R,N)
        Solver.storeInvariato = false;
        return;
    }
}
```

Questo è il caso non deterministico. Si aprono due casi, $lvar = m.getFirst()$ e $lvar \in^{(N-1)} m.sub()$ oppure $lvar \neq m.getFirst()$ e $lvar \in^{N} m.sub()$, questi due casi sono gestiti da uno switch, `switch (s.caseControl)`, se `s.caseControl == 0` si considera il primo caso, altrimenti si considera il secondo caso. Se `s.caseControl == 0`, si apre un punto di scelta, con lo stato dello `store` attuale, che andrà a considerare il caseControl 1

```
Solver.B.addChoicePoint(1, statoVarIn, statoStoreIn, domainStateIn);
```

Viene costruito il vincolo $lvar = m.getFirst()$ e aggiunto al solver con i comandi

```
Constraint s2 = new Constraint(lvar, Enviroment.eqCode, m.getFirst());
Solver.add(Solver.indexOf(s) + 1, s2);
```

Viene trattato il vincolo $lvar \in^{(N-1)} m.sub()$ con la chiamata a funzione

```
inN(s)
```

Se `s.caseControl == 1`, viene costruito il vincolo $lvar \neq m.getFirst()$ e aggiunto al solver con i comandi

```
Constraint s5 = new Constraint(lvar, Enviroment.neqCode, m.getFirst());
Solver.add(Solver.indexOf(s) + 1, s5);
```

Viene trattato il vincolo $lvar \in^{N} m.sub()$ con la chiamata a funzione

```
inN(s).
```

4.4 Vincolo di unione somma

Le regole di riscrittura del vincolo di **unione somma**, descritte nella tabella (3.3.2), sono implementate nel metodo `unionSum` della classe `RewritingConstraintsRules`. La testata del metodo ha il seguente aspetto:

```
private void unionSum(LBagProtected U, LBagProtected X,
                      LBagProtected Y, Constraint s) throws Failure;
```

dove U è l'**unione somma** tra X e Y. (`U.union(X,Y);`)
Il primo caso è

```
if(!X.isKnown() && !Y.isKnown() && !U.isKnown()) {
    return;
}
```

Infatti se X, Y e U non sono definiti (non-inizializzati) allora il vincolo è irriducibile, rimane invariato e si esce dalla funzione con il comando `return`.

Caso 3.13

```
if(U.isEmptySL()){ // se U = vuoto => X= vuoto e Y= vuoto
    Constraint s2 = new Constraint(Y, Enviroment.eqCode, ConcreteLBag.empty);
    Solver.add(Solver.indexOf(s) + 1, s2);
    s.arg1 = X;
    s.arg2 = ConcreteLBag.empty;
    s.arg3 = null;
    s.arg4 = null;
    eq(s);
    Solver.storeInvariato = false;
    return;
}
```

Se U è vuoto viene costruito il vincolo $Y = \emptyset$ con l'istruzione

```
Constraint s2 = new Constraint(Y, Enviroment.eqCode, ConcreteLBag.empty);
```

Viene aggiunto il vincolo al Solver con l'istruzione

```
Solver.add(Solver.indexOf(s) + 1, s2);
```

Viene trattato il vincolo $X = \emptyset$ con la chiamata a funzione

```
eq(s).
```

Caso 3.14

```
if(Y.isEmptySL()){ // se Y= vuoto => U = X
    s.arg1 = X;
    s.arg2 = U;
    s.arg3 = null;
    s.arg4 = null;
    eq(s);
    return;
}
```

Se Y è vuoto viene trattato il vincolo $U = X$ con la chiamata a funzione

```
eq(s).
```

Caso 3.15

```
if(X.isEmptySL()){ // se X= vuoto => U = Y
    s.arg1 = Y;
    s.arg2 = U;
    s.arg3 = null;
    s.arg4 = null;
    eq(s);
    return;
}
```

Se X è vuoto viene trattato il vincolo $U = Y$ con la chiamata a funzione

`eq(s).`

Caso 3.16

```
if(Y.isKnown() && Y.size() >= 1){ // Y = {{y|S}}
    LBag R = new ConcreteLBag();
    Constraint s2 = new Constraint(U, Enviroment.eqCode,
                    R.ins(Y.getFirst())); // U = {{y|R}}
    Solver.add(Solver.indexOf(s) + 1, s2);

    s.arg1 = X;
    s.arg2 = Y.sub(); //S
    s.arg3 = R;
    s.arg4 = null;
    union(s); // union(X,S,R)
    Solver.storeInvariato = false;
    return;
}
```

Se Y ha almeno un elemento inizializzato, viene costruito il vincolo $U = \{\!\{ y|R \}\!\}$ con l'istruzione

`Constraint s2 = new Constraint(U, Enviroment.eqCode, R.ins(Y.getFirst()));`

Viene aggiunto il vincolo al Solver con l'istruzione

`Solver.add(Solver.indexOf(s) + 1, s2);`

Viene trattato il vincolo $R = X \uplus Y.sub()$ (dove $Y.sub()$ significa il multi-insieme Y privato del primo elemento) con la chiamata a funzione

`union(s);`.

Caso 3.17

```
if(X.isKnown() && X.size() >= 1){ // X = {{x|S}}
```

```
    LBag R = new ConcreteLBag();
    Constraint s2 = new Constraint(U, Enviroment.eqCode,
                    R.ins(X.getFirst())); // U = {{x|R}}
    Solver.add(Solver.indexOf(s) + 1, s2);

    s.arg1 = X.sub(); // S
    s.arg2 = Y;
    s.arg3 = R;
    s.arg4 = null;
    union(s);
    Solver.storeInvariato = false;
    return;
}
```

Se X ha almeno un elemento inizializzato, viene costruito il vincolo $U = \{\!\{ x|R \}\!\}$ con il comando

```
Constraint s2 = new Constraint(U, Enviroment.eqCode, R.ins(X.getFirst()));
```

Viene aggiunto il vincolo al Solver con il comando

```
Solver.add(Solver.indexOf(s) + 1, s2);
```

Viene trattato il vincolo $R = X.sub() \uplus Y$ con la chiamata a funzione

```
union(s);.
```

Caso 3.18

```
if(U.size() >= 1) // caso: (X,Y,{{u|R}})
switch (s.caseControl) { // X= {{u|R2}} && union(R2,Y,R)
case 0:
    VarState statoVarIn = Solver.B.getVarState();
    StoreState statoStoreIn = Solver.B.getStoreState(s);
    Hashtable<ConcreteLVar, Interval> domainStateIn = Solver.B.getDomainState();
```

```
    Solver.B.addChoicePoint(1, statoVarIn, statoStoreIn, domainStateIn);

    LBag R2 = new ConcreteLBag();
    Constraint s2 = new Constraint(X, Enviroment.eqCode, R2.ins(U.getFirst()));
    Solver.add(Solver.indexOf(s) + 1, s2);

    s.arg1 = R2;
    s.arg2 = Y;
    s.arg3 = U.sub();
    s.arg4 = null;
    union(s);
    Solver.storeInvariato = false;
    return;

case 1: // Y= {{u|R3}} && union(X,R3,R)

    s.caseControl = 0;

    LBag R3 = new ConcreteLBag();
    Constraint s3 = new Constraint(Y, Enviroment.eqCode, R3.ins(U.getFirst()));
    Solver.add(Solver.indexOf(s) + 1, s3);

    s.arg1 = X;
    s.arg2 = R3;
    s.arg3 = U.sub();
    s.arg4 = null;
    union(s);
    Solver.storeInvariato = false;
    return;
}
```

Questo è il caso non deterministico. Se $U = \{\!\{ u|R \}\!\}$ si aprono due casi, $X = \{\!\{ u|R2 \}\!\}$ oppure $Y = \{\!\{ u|R3 \}\!\}$, questi due casi sono gestiti da uno switch, `switch (s.caseControl)`, se `s.caseControl == 0` si considera il caso $X = \{\!\{ u|R2 \}\!\}$, altrimenti si considera il caso $Y = \{\!\{ u|R3 \}\!\}$. Se `s.caseControl == 0`, si apre un punto di scelta, con lo stato dello `store` attuale, che andrà a considerare il caseControl 1

```
Solver.B.addChoicePoint(1, statoVarIn, statoStoreIn, domainStateIn);.
```

Viene costruito il vincolo $X = \{\!\{ u|R2 \}\!\}$ e aggiunto al solver con i comandi

```
Constraint s2 = new Constraint(X, Enviroment.eqCode, R2.ins(U.getFirst()));
Solver.add(Solver.indexOf(s) + 1, s2);
```

Viene trattato il vincolo $U.sub() = R2 \uplus Y$ con la chiamata a funzione

```
union(s)
```

Se `s.caseControl == 1`, viene costruito il vincolo $Y = \{\!\{ u|R3 \}\!\}$ e aggiunto al solver con i comandi

```
Constraint s3 = new Constraint(Y, Enviroment.eqCode, R3.ins(U.getFirst()));
Solver.add(Solver.indexOf(s) + 1, s3);
```

Viene trattato il vincolo $U.sub() = R3 \uplus X$ con la chiamata a funzione

```
union(s).
```

4.5 Vincolo di disgiunzione

Le regole di riscrittura del vincolo di **disgiunzione**, descritte nella tabella (3.3.3), sono state implementate nel metodo `disj` della classe `RewritingConstraintsRules`. Nella libreria JSetL era già implementato l'algoritmo che calcolava il vincolo di disgiunzione per gli **insiemi logici**. In questo caso la tabella delle regole di riscrittura relativa agli insiemi è identica a quella per i multi-insiemi.

L'algoritmo ha la stessa logica, si può quindi sfruttare la nuova interfaccia `LCollectionProtected`, per implementare un unico metodo che vada bene sia per gli insiemi che per i multi-insiemi. La testata di questo metodo avrà la seguente forma

```
protected void disj(LCollectionProtected set1, LCollectionProtected set2,
                    Constraint s) throws Failure {
```

dove `set1` e `set2` sono i due insiemi (o multi-insiemi) di cui si vuol testare la disgiunzione (`set1.disj(set2);`).

Caso di vincolo irriducibile

```
if (!set1.isKnown() && !set2.isKnown()) {
    return;
}
```

Caso 3.19

```
else if (set1.isEmptySL() || set2.isEmptySL()) {
    s.bool = true;
    return;
}
```

Caso 3.21

```
else if (set1.equals(set2)) {
    s.arg1 = set1;
    s.arg2 = set1.getEmpty();
    s.arg3 = null;
    s.cons = Enviroment.eqCode;
    eq(s);
    return;
}
```

Caso 3.22 e 3.23

```
else if (set1.isKnown() && !set2.isKnown()) {
    s.arg1 = set1.sub(); // s1||X
    Constraint st = new Constraint(set1.getFirst(),
              Enviroment.ninCode, set2);
    Solver.add(st);
    Solver.storeInvariato = false;
} else if (!set1.isKnown() && set2.isKnown()) {
    s.arg2 = set2.sub(); // X||s2
    Constraint se = new Constraint(set2.getFirst(),
              Enviroment.ninCode, set1);
    Solver.add(se);
    Solver.storeInvariato = false;
}
```

Caso 3.24

```
else {
    s.arg1 = set1.sub();
    s.arg2 = set2.sub();
    Constraint s1 = new Constraint(set1.getFirst(),
               Enviroment.neqCode, set2.getFirst());
    Constraint s2 = new Constraint(set1.getFirst(),
               Enviroment.ninCode, set2.sub());
    Constraint s3 = new Constraint(set2.getFirst(),
               Enviroment.ninCode, set1.sub());
    Solver.add(s1);
    Solver.add(s2);
    Solver.add(s3);
    Solver.storeInvariato = false;
}
```

4.6 Implementazione dei vincoli ricavati

Grazie ai lemmi 1, 2, 3 e 4, si possono implementare i vincoli che rappresentano le operazioni di $\subseteq$, $||$, $\cap$ e $\cup$ come semplici congiunzioni degli altri vincoli. Dalla definizione di vincolo 3.12 sappiamo che una congiunzione di vincoli è un vincolo.

4.6.1 Vincolo di sottoinsieme

Per implementare il vincolo di sottoinsieme si segue la regola 3.37. L'implementazione del vincolo è la seguente

```
public Constraint subset(LCollection s) {
    LBagProtected X = new ConcreteLBag();
    return new Constraint(Enviroment.unionCode, this, X, s);
}
```

4.6.2 Vincolo di intersezione

Per implementare il vincolo di intersezione si segue la regola 3.35. L'implementazione del vincolo è la seguente

```
public ConstraintsConjunction inters(LCollection r, LCollection s) {
     ConstraintsConjunction v = new ConstraintsConjunction();
     LBagProtected R = new ConcreteLBag();
     LBagProtected S = new ConcreteLBag();
     v.add(r.union(R,this));
     v.add(s.union(S,this));
     v.add(R.disj(S));
     return v;
 }
```

Il vincolo è costruito a partire da una congiunzione logica di vincoli di unione e disgiunzione.

4.6.3 Vincolo di differenza

Per implementare il vincolo di differenza si segue la regola 3.36. L'implementazione del vincolo è la seguente

```
public ConstraintsConjunction differ(LCollection r, LCollection s) {
            ConstraintsConjunction v = new ConstraintsConjunction();
            LBagProtected Z = new ConcreteLBag();
            LBagProtected E = new ConcreteLBag();
            v.add(Z.inters(r,s));
            v.add(r.union(this,E));
            v.add(Z.eq(E));
            return v;
}
```

4.6.4 Vincolo di unione massimo

Per implementare il vincolo di unione massimo si segue la regola 3.38. L'implementazione del vincolo è la seguente

```
public ConstraintsConjunction unionMax(LBag r, LBag s) {
            ConstraintsConjunction v = new ConstraintsConjunction();
            LBagProtected W = new ConcreteLBag();
            LBagProtected X = new ConcreteLBag();
            LBagProtected E1 = new ConcreteLBag();
            LBagProtected E2 = new ConcreteLBag();
            v.add(X.union(r,s));
            v.add(W.inters(r,s));
            v.add(X.eq(E1));
            v.add(W.eq(E2));
            v.add(this.differ(E1,E2));
            return v;
}
```

4.7 Un esempio

Si vuole ora fare un esempio dell'utilizzo, in JSetL, del vincolo di `unione somma`, per calcolare le soluzioni dell'esempio (3.4).

```
int[] array = {1,1};
LBag U = new ConcreteLBag("U", array); // U = +{1,1}+
LBag X = new ConcreteLBag("X");
LBag Y = new ConcreteLBag("Y");

Solver.add(U.union(X,Y)); // U = X U Y
Solver.solve();
```

Quando viene invocato il metodo `solve()` viene avviato l'algoritmo di risoluzione dei vincoli. Viene trovato il vincolo `U.union(X,Y)` nel Constraint Store e viene invocato il metodo `unionSum` della classe `RewritingConstraintsRules`. L'algoritmo di risoluzione continua ad applicare le regole fino a che i vincoli non rimangono inviariati; questi saranno le soluzioni. In questo esempio il programma darà in output (la notazione `+{...}+` indica un multi-insieme)

```
-------- SOLUTIONS --------
U = +{1,1}+

X = +{1,1}+

Y = +{}+
```

```
-------- SOLUTIONS --------
U = +{1,1}+

X = +{1}+

Y = +{1}+

-------- SOLUTIONS --------

U = +{1,1}+

X = +{}+

Y = +{1,1}+
```

Come si può vedere le soluzioni sono le stesse dell'esempio 3.4.

Capitolo 5

Conclusioni e lavori futuri

Nella prima parte di questa tesi abbiamo progettato e realizzato le interfacce che permettono, a livello utente (interfacce pubbliche) o internamente al package (interfacce protected), di unire i concetti comuni fra variabili logiche, insiemi, liste e multi-insiemi, mantenendo la compatibilità con JSetL 1.3.2. Abbiamo creato le interfacce `LVar`, `LCollection`, `LSet`, `LBag` ed `LLst` per l'utilizzo della libreria da parte dell'utente. Abbiamo creato le interfacce `LVarProtected`, `LCollectionProtected`, `LSetProtected`, `LBagProtected` ed `LLstProtected` per la gestione dell'implementazione interna al package. Con questa nuova implementazione possiamo sfruttare tutti i vantaggi derivati dall'ereditarieta delle classi. Le future implementazioni, fatte per migliorare le prestazioni o per implementare classi con strategie diverse, saranno già utilizzabili immediatamente dall'utente grazie alle interfacce. Invece le nuove implementazioni interne al package potranno sfruttare le interfacce protected per poter creare metodi che abbiano logiche comuni alle collezioni logiche o alle variabili logiche, come è stato fatto nell'implementazione (4.5).

Un lavoro futuro potrà consistere nell'implementare le **variabili logiche intere** che possono essere rappresentate dall'interfaccia `LVarInt`, che estende l'interfaccia `LVar` e dovrà essere compatibile con l'interfaccia `LvarExpr`. La classe `ConcreteLVarInt` implementerà l'interfaccia.

Un'altra possibilità di lavoro futuro consiste, come suggerisce la tesi di Filippi [1], nel creare l'astrazione `MutableLst` che permetta di operare contemporaneamente con metodi delle liste `java.util.List` e delle liste `LLst`. Nella seconda parte di questa tesi abbiamo esteso a livello teorico alcuni aspetti della teoria dei multi-insiemi. Abbiamo dato la definizione formale di multi-insieme e delle operazioni insiemistiche di appartenenza multipla, unione somma, disgiunzione, intersezione, sottoinsieme, differenza ed unione massimo. Abbiamo dimostrato che per ognuna di queste operazioni esiste una congiunzione di operazioni, avente solo unione somma e disgiunzione, equivalente. Usando questi risultati abbiamo implementato in JSetL i vincoli di appartenenza multipla, unione somma, disgiunzione, intersezione, sottoinsieme, differenza ed unione massimo.
Lavori futuri possono consistere nel dimostrare formalmente che le regole di riscrittura dei vincoli, ricavate dalle definizioni delle operazioni sui multi-insiemi, rispettano le definizioni formali e dimostrare che le congiunzioni di vincoli generate al termine del processo di riscrittura dei vincoli unione somma e disgiunzione siano sempre soddisfacibili. Altri lavori futuri possono concentrarsi sull'estendere anche la teoria relativa alle **liste logiche** implementandone i vincoli in JSetL.

Appendice A

Implementazione delle interfacce di JSetL

Di seguito sono elencati i metodi delle interfacce relative alla trattazione uniforme di **variabili logiche, insiemi, liste** e **multi-insiemi**.

A.1 Interfaccia LVar

L'interfaccia `LVar` è un'interfaccia utente e può essere utilizzata dall'utente per gestire una qualsiasi istanza di classe che sia una implementazione di `LVar`. I metodi rappresenteranno quindi le operazioni comuni alle **variabili logiche**.

metodo	descrizione
`getValue()`	`Restituisce il valore della variabile logica.`
`isKnown()`	`Restituisce true se la variabile logica è inizializzata, false altrimenti.`
`getName()`	`Restituisce il nome della variabile logica.`

metodo	descrizione
`isGround()`	Restituisce true se la variabile logica è ground, false altrimenti. Ground vuole dire che la variabile logica è completamente specificata. Nel caso di una collezione logica anche tutti gli elementi devono essere specificati.
`output()`	Rappresenta in formato output la variabile logica.
`equals(Object L)`	Restituisce true se la variabile logica è uguale (logicamente) ad L, false altrimenti.
`print()`	Rappresenta in stringa la variabile logica e la stampa in output.
`toString()`	Rappresenta in stringa la variabile logica.
`eq(Object z)`	Costruisce e restituisce il vincolo di uguaglianza.
`neq(Object z)`	Costruisce e restituisce il vincolo di disuguaglianza.
`in(LCollection z)`	Costruisce e restituisce il vincolo di appartenenza.
`in(LCollection z, LVar N);`	Costruisce e restituisce il vincolo di appartenenza multipla. Nel caso z non sia un LBag viene considerato come il vincolo di appartenenza semplice in(LCollection z).
`nin(LCollection z)`	Costruisce e restituisce il vincolo di non appartenenza.
`eq(LvarExpr a)`	Costruisce e restituisce una congiunzione di vincoli di uguaglianza.

A.2 Interfaccia LVarProtected

L'interfaccia `LVarProtected` è un'interfaccia privata e può essere utilizzata nell'implementazione per gestire, in modo privato, una qualsiasi istanza di classe che sia una implementazione di `LVarProtected`. I metodi rappresenteranno quindi le operazioni comuni, utilizzate internamente al package, delle **variabili logiche**.
`LVarProtected` estende `LVar`.

metodo	descrizione
`setVal(Object v)`	`Modifica il valore della variabile logica, salvandoci il valore v.`
`getId()`	`Restituisce l'identificativo della variabile logica.`
`isInit()`	`Restituisce true se la variabile logica è inizializzata, false altrimenti.`
`isInit(boolean init)`	`Modifica il valore booleano init della variabile logica, salvandoci il valore init.`
`setId()`	`Modifica l'identificativo della variabile logica, salvandoci il valore preso da un contatore statico.`
`setId(int id)`	`Modifica l'identificativo della variabile logica, salvandoci il valore id.`
`setEqu(LVarProtected equ)`	`Modifica il puntatore che punta alla variabile logica uguale a questa, salvandoci il puntatore equ.`

metodo	descrizione
`getEqu()`	`Restituisce il puntatore che punta alla variabile logica uguale a questa.`
`setName(String name)`	`Modifica il nome della variabile, salvandoci il valore name.`
`clone()`	`Costruisce e restituisce una variabile logica uguale a questa.`

A.3 Interfaccia LCollection

L'interfaccia `LCollection` è un'interfaccia utente e può essere utilizzata dall'utente per gestire una qualsiasi istanza di classe che sia una implementazione di `LCollection`. I metodi rappresenteranno quindi le operazioni comuni alle **collezioni logiche**.
`LCollection` estende `LVar`.

metodo	descrizione
`getEmpty()`	`Restituisce la collezione vuota.`
`getList()`	`Restituisce il vettore degli elementi specificati della collezione logica.`
`getRest()`	`Restituisce la collezione logica che rappresenta il resto.`
`isEmpty()`	`Restituisce true se la collezione logica è la collezione vuota, false altrimenti. Se la collezione logica è non-inizializzata viene lanciata l'eccezione NotInitVarException.`
`isBound()`	`Restituisce true se la collezione logica è bound, false altrimenti. Bound significa che la collezione non ha resto.`

metodo	descrizione
`occurs(Object x)`	Restituisce true se x occorre nella collezione logica, false altrimenti.
`size()`	Restituisce il numero degli elementi specificati della collezione logica.
`get(int i)`	Restituisce l'elemento in posizione i nel vettore degli elementi specificati della collezione logica.
`getFirst()`	Restituisce il primo elemento degli elementi specificati della collezione logica.
`ins(int n)`	Costruisce una collezione logica uguale a questa, inserisce n e restituisce la collezione risultante.
`ins(Object v)`	Costruisce una collezione logica uguale a questa, inserisce v e restituisce la collezione risultante.
`insAll(int[] arr)`	Costruisce una collezione logica uguale a questa, inserisce tutti gli elementi dell'array arr e restituisce la collezione risultante.
`insAll(LVar[] arr)`	Costruisce una collezione logica uguale a questa, inserisce tutti gli elementi dell'array arr e restituisce la collezione risultante.
`toVector()`	Restituisce un vettore di tutti gli elementi specificati della collezione andando a cercare ricorsivamente nel resto.

metodo	descrizione
`toVector(Vector v)`	Restituisce un vettore, concatenato al vettore v, di tutti gli elementi specificati della collezione andando a cercare ricorsivamente nel resto.
`disj(LCollection set)`	Costruisce e restituisce il vincolo di disgiunzione.
`ndisj (LCollection set)`	Costruisce e restituisce il vincolo di non-disgiunzione.
`union(LCollection set1, LCollection set2)`	Costruisce e restituisce il vincolo di unione.
`nunion(LCollection set1, LCollection set2)`	Costruisce e restituisce il vincolo di non-unione.
`size(Integer N)`	Costruisce e restituisce il vincolo di size.
`size(LVar X)`	Costruisce e restituisce il vincolo di size.

metodo	descrizione
`subset(LCollection s)`	Costruisce e restituisce il vincolo di sottoinsieme.
`nsubset(LCollection s)`	Costruisce e restituisce il vincolo di non-sottoinsieme.
`differ(LCollection r, LCollection s)`	Costruisce e restituisce il vincolo di differenza.
`ndiffer(LCollection r, LCollection s)`	Costruisce e restituisce il vincolo di non-differenza.
`inters(LCollection r, LCollection s)`	Costruisce e restituisce il vincolo di intersezione.
`ninters(LCollection r, LCollection s)`	Costruisce e restituisce il vincolo di non-intersezione.

A.4 Interfaccia LCollectionProtected

L'interfaccia `LCollectionProtected` è un'interfaccia privata e può essere utilizzata nell'implementazione per gestire, in modo privato, una qualsiasi istanza di classe che sia una implementazione di `LCollectionProtected`. I metodi rappresenteranno quindi le operazioni comuni, utilizzate interna-

mente al package, delle **collezioni logiche**.

`LCollectionProtected` estende `LCollection` e `LVarProtected`.

metodo	descrizione
`setRest(LCollectionProtected rest)`	`Modifica il resto della collezione logica, salvandoci il valore rest.`
`setList(Vector list)`	`Modifica il vettore di elementi specificati della collezione logica, salvandoci il valore list.`
`isEmptySL()`	`Restituisce true se la collezione logica è la collezione vuota, false altrimenti. Se la collezione è non-inizializzata restituisce false.`
`isInterval()`	`Restituisce true se la collezione è un intevallo, false altrimenti.`
`isInterval(boolean isInterval)`	`Setta la collezione ad intervallo se isInterval = true.`

metodo	descrizione
`getInterval()`	`Restituisce l'intervallo.`
`setInterval(Interval inter)`	`Modifica l'intervallo, salvandoci il valore inter.`
`sub()`	`Costruisce e restituisce una collezione logica uguale a questa ma senza il primo elemento.`
`sub(Object z)`	`Costruisce e restituisce una collezione logica uguale a questa ma senza l'elemento z.`
`subFirst()`	`Costruisce e restituisce una collezione logica uguale a questa ma senza il primo elemento. Con l'ipotesi che la collezione non è vuota.`

A.5 Interfaccia LSet

L'interfaccia `LSet` è un'interfaccia utente e può essere utilizzata dall'utente per gestire una qualsiasi istanza di classe che sia una implementazione di `LSet`. I metodi rappresenteranno quindi le operazioni comuni agli **insiemi logici**.

`LSet` estende `LCollection`.

metodo	descrizione
`ins(int n)`	Costruisce un insieme logico uguale a questo, inserisce n e restituisce l'insieme logico risultante.
`ins(Object v)`	Costruisce un insieme logico uguale a questo, inserisce v e restituisce l'insieme logico risultante.
`insAll(int[] arr)`	Costruisce un insieme logico uguale a questo, inserisce tutti gli elementi, di tipo int, dell'array arr e restituisce l'insieme logico risultante.
`insAll(LVar[] arr)`	Costruisce un insieme logico uguale a questo, inserisce tutti gli elementi, di tipo LVar, dell'array arr e restituisce l'insieme logico risultante.
`getRest()`	Restituisce l'insieme logico che rappresenta il resto.
`normalize()`	Costruisce e restituisce l'insieme logico normalizzato.
`read()`	Costruisce e restituisce un insieme logico basandosi sulla sintassi ricevuta da input.
`setName(String n)`	Modifica il nome, salvandoci n.
`allDifferent()`	Costruisce e restituisce il vincolo di differenza degli elementi.

A.6 Interfaccia LSetProtected

L'interfaccia `LSetProtected` è un'interfaccia privata e può essere utilizzata nell'implementazione per gestire, in modo privato, una qualsiasi istanza di classe che sia una implementazione di `LSetProtected`. I metodi rappre-

senteranno quindi le operazioni comuni, utilizzate internamente al package, degli **insiemi logici**.

`LSetProtected` estende `LSet` e `LCollectionProtected`.

metodo	descrizione
`getEqu()`	`Restituisce il puntatore che punta all'insieme logico uguale a questo.`
`clone()`	`Costruisce e restituisce un insieme logico uguale a questo.`
`concat(LSetProtected set)`	`Costruisce e restituisce un insieme logico uguale a questo concatenato a set.`
`sost(ConcreteLVar y, ConcreteLVar x)`	`Costruisce e restituisce un insieme logico uguale a questo, con il valore x al posto del valore y.`

metodo	descrizione
`sub()`	`Costruisce e restituisce un insieme logico uguale a questo ma senza il primo elemento.`
`sub(Object z)`	`Costruisce e restituisce un insieme logico uguale a questo ma senza l'elemento z.`
`subFirst()`	`Costruisce e restituisce un insieme logico uguale a questo ma senza il primo elemento. Con l'ipotesi che l'insieme non è vuoto.`

A.7 Interfaccia LBag

L'interfaccia `LBag` è un'interfaccia utente e può essere utilizzata dall'utente per gestire una qualsiasi istanza di classe che sia una implementazione di `LBag`. I metodi rappresenteranno quindi le operazioni comuni ai **multi-insiemi logici**.

`LBag` estende `LCollection`.

metodo	descrizione
`untail()`	`Restituisce un vettore di tutti gli elementi specificati del multi-insieme andando a cercare ricorsivamente nel resto.`
`ins(int n)`	`Costruisce un multi-insieme logico uguale a questo, inserisce n, e restituisce il multi-insieme logico risultante.`
`ins(Object v)`	`Costruisce un multi-insieme logico uguale a questo, inserisce v, e restituisce il multi-insieme logico risultante.`

metodo	descrizione
`insAll(int[] arr)`	`Costruisce un multi-insieme logico uguale a questo, inserisce tutti gli elementi dell'array arr di tipo int e restituisce il multi-insieme logico risultante.`
`insAll(LVar[] arr)`	`Costruisce un multi-insieme logico uguale a questo, inserisce tutti gli elementi dell'array arr di tipo LVar e restituisce il multi-insieme logico risultante.`
`getRest()`	`Restituisce il multi-insieme logico che rappresenta il resto.`

A.8 Interfaccia LBagProtected

L'interfaccia `LBagProtected` è un'interfaccia privata e può essere utilizzata nell'implementazione per gestire, in modo privato, una qualsiasi istanza di classe che sia una implementazione di `LBagProtected`. I metodi rappresenteranno quindi le operazioni comuni, utilizzate internamente al package, ai **multi-insiemi logici**.

`LBagProtected` estende `LBag` e `LCollectionProtected`.

metodo	descrizione
`sub()`	`Costruisce e restituisce un multi-insieme logico uguale a questo ma senza il primo elemento.`
`sub(Object z)`	`Costruisce e restituisce un multi-insieme logico uguale a questo ma senza l'elemento z.`
`subFirst()`	`Costruisce e restituisce un multi-insieme logico uguale a questo ma senza il primo elemento. Con l'ipotesi che il multi-insieme non è vuoto.`
`getEqu()`	`Restituisce il puntatore che punta al multi-insieme logico uguale a questo.`
`clone()`	`Costruisce e restituisce un multi-insieme logico uguale a questo.`
`concat(LBagProtected bag)`	`Costruisce e restituisce un multi-insieme logico uguale a questo concatenato a bag.`

A.9 Interfaccia LLst

L'interfaccia `LLst` è un'interfaccia utente e può essere utilizzata dall'utente per gestire una qualsiasi istanza di classe che sia una implementazione di `LLst`. I metodi rappresenteranno quindi le operazioni comuni alle **liste logiche**.

metodo	descrizione
`ins(int n)`	Costruisce una lista logica uguale a questa, inserisce n e restituisce la lista risultante.
`ins(Object v)`	Costruisce una lista logica uguale a questa, inserisce v e restituisce la lista risultante.
`insAll(int[] arr)`	Costruisce una lista logica uguale a questa, inserisce tutti gli elementi dell'array arr di tipo int e restituisce la lista logica risultante.
`insAll(LVar[] arr)`	Costruisce una lista logica uguale a questa, inserisce tutti gli elementi dell'array arr di tipo LVar e restituisce la lista logica risultante.
`getRest()`	Restituisce la lista logica che rappresenta il resto.

metodo	descrizione
`insn(Object ob)`	Costruisce una lista logica uguale a questa, nel vettore degli gli elementi specificati inserisce in fondo ob e restituisce la lista risultante.
`insn(int n)`	Costruisce una lista logica uguale a questa, nel vettore degli gli elementi specificati inserisce in fondo n e restituisce la lista risultante.
`insnAll(int[] arr)`	Costruisce una lista logica uguale a questa, nel vettore degli elementi specificati inserisce in fondo tutti gli elementi dell'array arr di tipo int e restituisce la lista logica risultante.
`insnAll(Object[] arr)`	Costruisce una lista logica uguale a questa, nel vettore degli elementi specificati inserisce in fondo tutti gli elementi dell'array arr di tipo LVar e restituisce la lista logica risultante.
`normalize()`	Costruisce una lista logica normalizzata e la restituisce.

A.10 Interfaccia LLstProtected

L'interfaccia `LLstProtected` è un'interfaccia privata e può essere utilizzata nell'implementazione per gestire, in modo privato, una qualsiasi istanza di classe che sia una implementazione di `LLstProtected`. I metodi rappresen-

teranno quindi le operazioni comuni, utilizzate internamente al package, alle **liste logiche**.

`LLstProtected` estende `LLst` e `LCollectionProtected`.

metodo	descrizione
`getEqu()`	`Restituisce il puntatore che punta alla lista logica uguale a questa.`
`clone()`	`Costruisce e restituisce una lista logica uguale a questa.`
`sost(ConcreteLVar y, ConcreteLVar x)`	`Costruisce e restituisce una lista logica uguale a questa, con il valore x al posto del valore y.`
`concat(LVarProtected l)`	`Costruisce e restituisce una lista logica uguale a questa concatenata con l.`

metodo	descrizione
`sub()`	`Costruisce e restituisce una lista logica uguale a questa ma senza il primo elemento.`
`sub(Object z)`	`Costruisce e restituisce una lista logica uguale a questa ma senza l'elemento z.`
`subFirst()`	`Costruisce e restituisce una lista logica uguale a questa a questo ma senza il primo elemento. Con l'ipotesi che la lista logica non è una lista vuota.`

Bibliografia

[1] Michele Giacomo Filippi
Un'interfaccia uniforme per la programmazione con insiemi e vincoli insiemistici in Java
Tesi di laurea triennale, Università di Parma, 2008

[2] Agostino Dovier, Carla Piazza, Enrico Pontelli, Gianfranco Rossi
Sets and Constraint Logic Programming
ACM Transaction on Programming Languages and Systems, 2000

[3] Agostino Dovier, Carla Piazza, Enrico Pontelli, Gianfranco Rossi
A uniform approach to constraint-solving for lists, multisets, compact lists, and sets
ACM Transactions on Computational Logic, 2008

[4] D. Singh , A. M. Ibrahim , T. Yohanna and J. N. Singh
AN OVERVIEW OF THE APPLICATIONS OF MULTISETS
AMS Mathematics Subject Classication, 2000

[5] JSetL Homepage
http://www.math.unipr.it/ gianfr/JSetL/index.html

[6] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software-Practice and Experience, 2006

[7] Sun Microsystems Java SE Homepage
http://java.sun.com/javase/