UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE

MATEMATICHE, FISICHE e NATURALI

Corso di Laurea in Informatica

Tesi di Laurea Triennale

# Un'interfaccia uniforme per la programmazione con insiemi e vincoli insiemistici in Java

Candidato:

**Michele Giacomo Filippi**

Relatore:

**Prof. Gianfranco Rossi**

Anno Accademico 2007/2008

*Ai miei genitori,*
*ai miei nonni,*
*alla mia fidanzata,*
*e a tutti i miei amici*

# Indice

# Introduzione

In informatica un **insieme** è una struttura dati che consiste in una *collezione di oggetti disposti senza un particolare ordine e senza oggetti ripetuti* e corrisponde al concetto matematico di insieme finito.
L'uso di tale astrazione, nella progettazione e sviluppo del software, può accrescere la produttività e la velocità del programmatore nonchè migliorare la chiarezza e la leggibilità del codice, a scapito di una perdita in efficienza.

Interessanti esempi di uso di insiemi nei linguaggi di programmazione si hanno nel linguaggio procedurale SETL [3], basato sulla teoria degli insiemi, dove gli insiemi sono oggetti primitivi del linguaggio. È comunque caratteristica comune di tutti i più diffusi linguaggi di programmazione offrire un qualche supporto, seppure spesso limitato, agli insiemi.

Nel loro uso comune, dai linguaggi di programmazione ad oggetti, come C++, con le classi di librerie standard `set` o `hash_set`, e Java [7], con l'interfaccia `Set` e le sue implementazioni `HashSet` o `TreeSet`, gli insiemi hanno caratteristiche comuni: sono sempre specificati e limitati e nella risoluzione di un problema che li coinvolge si eseguono operazioni passo-passo, come inserimento e rimozione di elementi, per ottenere la soluzione. Chiameremo questo tipo di insiemi "insiemi standard".

Un diverso approccio si ha con gli "insiemi logici" nella programmazione a vincoli con dominio gli insiemi (come in CLP($\mathcal{SET}$)[2] e JSetL [5]); gli

insiemi possono anche essere parzialmente specificati e i vincoli si limitano a imporre le proprietà di cui deve essere dotata la soluzione da trovare senza precisare le azioni da compiere.

Il lavoro di tesi proposto continua e completa il lavoro di tesi di Delia Di Giorgio [1]. L'**obiettivo** generale è *realizzare un'interfaccia uniforme per la programmazione con insiemi in* JSetL *che racchiuda le due concezioni, gli* "insiemi standard" Java *e gli* "insiemi logici" JSetL.

L'interfaccia deve permettere di operare sia con i metodi forniti dagli insiemi Java sia con quelli degli insiemi JSetL. La sua implementazione deve operare sugli "insiemi standard" comunque creati nel modo più efficiente e deve lasciare inalterato il package JSetL. Il tutto deve essere corredato da una completa documentazione utente.

La tesi è organizzata come segue.
Nel primo capitolo sono presentati la nozione di insieme e una breve panoramica su SETL.
Il secondo capitolo mostra le classi degli insiemi in Java, che saranno utilizzate in seguito.
Nel terzo capitolo si ha una breve introduzione a JSetL, con particolare attenzione alle classi che permettono di realizzare gli insiemi in JSetL.
Il quarto capitolo è dedicato alla visione utente dell'interfaccia unica per la gestione degli insiemi in JSetL e all'importanza di una efficace documentazione.
Nel quinto capitolo sono descritte la progettazione e l'implementazione dell'interfaccia.
Il sesto capitolo è dedicato ad alcuni esempi.
Infine seguono le conclusioni e considerazioni su eventuali lavori futuri.
In appendice si ha la documentazione in stile Javadoc dei metodi visibili all'utente dell'interfaccia.

# Capitolo 1

# Insiemi e programmazione con insiemi

Questo capitolo mostra l'importanza della nozione di insieme nella programmazione, a partire dal concetto intuitivo e da qualche richiamo dalla matematica fino agli insiemi in informatica e a SETL, un linguaggio di programmazione basato sulla teoria degli insiemi.

## 1.1 La nozione di insieme

Intuitivamente con il termine **insieme** si indica una *collezione di oggetti chiamati elementi dell'insieme*.
Ciò che distigue il concetto di insieme da altri concetti analoghi sono le seguenti proprietà:

- un elemento o fa parte di un insieme o non ne fa parte;
- un elemento non può comparire più di una volta in un insieme;
- gli elementi di un insieme non hanno ordine;
- gli elementi di un insieme lo caratterizzano univocamente.

Queste semplici proprietà, dedotte da come ci si aspetta sia un insieme e quindi dalla generalizzazione del concetto di insieme finito, in matemati-

ca sono formalizzate nella **teoria degli insiemi** (branca della matematica creata dal tedesco Georg Cantor).
Una caratteristica della teoria degli insiemi è che ogni oggetto che tratta è un insieme, gli elementi sono quindi definiti in termini di insiemi. Il concetto di appartenenza di un elemento $X$ a un insieme $A$, $X \in A$, è un concetto primitivo cioè viene utilizzato senza darne una definizione esplicita.
Sono riportati come esempio due degli assiomi studiati e accettati della teoria degli insiemi:

1. (Assioma di estensionalità), se due insiemi hanno gli stessi elementi essi sono uguali:

   $$\forall A \forall B(\forall X(X \in A \longleftrightarrow X \in B) \longrightarrow A = B)$$

   Per ogni insieme $A$ e per ogni insieme $B$, se per ogni $X$, $X$ appartiene ad $A$ se e solo se $X$ appartiene a $B$, allora $A = B$.

2. (Assioma dell'insieme vuoto), esiste un insieme privo di elementi:

   $$\exists A(\forall B \neg(B \in A))$$

   Esiste un insieme $A$ tale che per ogni $B$, $B$ non appartiene ad $A$.

## 1.2 Insiemi in informatica

Molti linguaggi di programmazione offrono la possilità di utilizzare oggetti simili agli insiemi, modellando il concetto matematico di insieme finito con oggetti di tipo `set`.

**Definizione 1.1** *Un* `set` *è una collezione finita di oggetti disposti senza un particolare ordine e senza oggetti ripetuti.*

Programmare con i `set` significa pensare e scrivere programmi ragionando in termini matematici insiemistici, cercando di sfruttare appieno le possibilità che tale approccio offre.
L'uso dei `set` è spesso frenato dalla minore efficienza rispetto all'uso di altre

strutture dati, ma spesso dove l'efficienza non è uno dei requisiti primari, ad esempio nella progettazione e sperimentazione di nuovi programmi, i `set` sono *un veloce e valido strumento per migliorare l'espressività, la chiarezza e la leggibilità* del codice.
Uno dei primi linguaggi a supportare i `set` è stato Pascal, linguaggio di programmazione procedurale sviluppato da Niklaus Wirth nel 1970, in cui i `set` sono un tipo di dati built-in. Oggi in C++ sono previste classi di librerie standard `set` o `hash_set`, in Java l'interfaccia `Set` e le sue implementazioni `HashSet` o `TreeSet`. Esistono anche linguaggi di programmazione basati sulla teoria degli insiemi: SETL ne è un esempio.

## 1.3 Il linguaggio di programmazione SETL

"SETL (for SET Language) is a high level general purpose language which allow a large variety of programming problems to be solved in an efficient manner with a minimum amount of effort."
(Robert B.K. Dewar) [3]

SETL è un linguaggio di programmazione basato sulla teoria degli insiemi. E' stato sviluppato originariamente da Jack Schwartz nei primi anni '70 al NYU Courant Institute of Mathematical Sciences.

SETL fornisce tipi di dato semplici:

- **integers**
  Rappresentano *valori interi*, illimitati.
- **floating-point numbers**
  Rappresentano *numeri reali*, quelli disponibili dalla macchina in uso.
- **character strings**
  Rappresentano *stringhe* di lunghezza arbitraria di caratteri, i caratteri sono quelli disponibili sulla particolare macchina in uso.

- **boolean values**
  La logica è a due valori: TRUE, FALSE.

- **atom**
  Speciali *valori unici* usati nella costruzione di map.

SETL fornisce tipi di dato composti:

- **unordered set**
  Rappresentano *insiemi finiti*.

- **tuple**
  Rappresentano *liste*.

- **map**
  Rappresentano *insiemi di coppie ordinate*.

Un particolare tipo di dato è `OM`: una costante che rappresenta un *valore indefinito*. Sotto certe circostanze, alcune computazioni possono terminare senza restituire un valore di tipo aspettato; ad esempio per operazioni che interessano elementi di un insieme, se questo insieme è vuoto, la computazione termina ma il valore risultante è indefinito. Più precisamente `OM` è restituito nelle seguenti circostanze: variabile non definita, elemento indefinito di una tupla, selezione di un elemento dall'insieme vuoto, e da particolari operatori applicati all'insieme o alla tupla vuoti.
I *valori di unordered set, tuples e maps* sono detti **elements**. Gli elements di insiemi e liste possono essere di qualsiasi tipo ad eccezione di `OM`, e quindi possono anche essere a loro volta unordered set, tuple e map.
Alle variabili in SETL può essere assegnato un valore, con il comando:

```
nome:=valore;
```

dove `nome` rappresenta il nome della variabile e `valore` il valore da assegnare. L'assegnamento è distruttivo. Diversamente da altri linguaggi di programmazione al nome della variabile non è associato il tipo di dati che

è invece determinato dall'ultimo valore assegnato. Nel comando di assegnamento `nome` deve iniziare per una lettera, può essere di lunghezza arbitraria e contenere lettere, cifre, e caratteri speciali.
Come nell'esempio del comando di assegnamento, ogni comando in SETL termina con punto e virgola, l'esecuzione è sequenziale.

La *sintassi* di SETL per quanto riguarda definizione e manipolazione di insiemi riproduce il linguaggio della *teoria degli insiemi*.
Si può denotare un unordered set con *notazione estensionale* elencando gli elementi tra parentesi graffe separati da virgola; l'insieme vuoto è un elemento SETL valido ed è denotato da {}. Analogamente, per le tuple si elencano gli elementi ordinati tra parentesi quadre. Una map è denotata come un unordered set i cui elements sono tuple di due elements, con la possibilià di accedere agli elementi come fosse un array associativo (specificando la chiave, la prima componente, tra parentesi tonde se la map è una funzione, graffe altrimenti).
È possibile denotare unordered set con *notazione intensionale* specificando la proprietà caratteristica degli elementi, attraverso quantificatori e test.
SETL permette di costruire espressioni booleane quantificate usando i quantificatori esistenziale e universale della logica dei predicati del primo ordine.
I test quantificati sono particolari espressioni che possono essere utilizzate in contesti che richiedono test:

```
exists iterator|test
notexists iterator|test
forall iterator|test
```

Nella valutazione di tali test sono implicitamente effettuati cicli. Ad esempio per in caso `exists` il risultato è TRUE se il test è TRUE per un qualsiasi elemento coinvolto, se un elemento è trovato l'iterazione termina, altrimenti se il test fallisce per tutti gli input il risultato è FALSE.

SETL fornisce *operazioni primitive* su unordered set, tuple e map, come ne-

gli insiemi l'unione, l'intersezione, il prodotto cartesiano e altre. Ad esempio siano `R`, `S` due unordered set, il comando:

```
C:=R+S;
```

dove `+` è l'operatore binario di *unione* di unordered set, assegna a `C` l'unione di `R` e `S`.
SETL fornisce inoltre iteratori per effettuare cicli sulle strutture dati. Ad esempio il costrutto semplificato per l'iterazione condizionata:

```
loop while Test do Com end;
```

ripeti finchè l'espressione `Test` è TRUE il comando `Com`, dove `Test` può essere un'espressione con operatori binari come uguale per il test di uguaglianza, minore e altri tra cui l'operatore di inclusione `in`, che è verificato se l'operando di sinistra è elemento dell'operando di destra, o di sottoinsieme `subset`, che è verificato se l'operando di sinistra è sottoinsieme dell'operando di destra.
SETL è quindi sufficientemente espressivo per codificare naturali illimitati e comprendende i comandi di base di assegnamento, esecuzione sequenziale ed iterazione condizionata, SETL è Turing-completo [4].

Una presentazione formale di SETL è rimandata alla bibliografia [3].

## 1.4 Esempi di programmazione con gli insiemi

Gli esempi in SETL mostrano l'importanza degli insiemi nella risoluzione di problemi; in questo capitolo si vedono due semplici esempi.
Il primo programma calcola il massimo di un insieme di valori interi. Da questo programma si può notare la dichiaratività del linguaggio, infatti non interessa trovare la sequenza di istruzioni necessaria per determinare l'elemento massimo dell'insieme, si definisce un sottoinsieme dell'insieme originario i cui elementi rispettano la proprietà dell'elemento massimo di un insieme

che è scritta nel linguaggio della teoria degli insiemi. La sua determinazione effettiva è lasciata all'interprete SETL.

```
A:={1,6,4,8,10,5};
P:={x in A | forall y in A | x >= y};
print(P);
```

Il frammento di programma produce il seguente output:

```
{10}
```

Nella prima dichiarazione è definito l'insieme `A` per elencazione, l'insieme `P` è invece definito con notazione intensionale, è l'insieme degli `x` appartenenti ad `A` tali che per ogni `y` appartenente ad `A`, `x` è maggiore o uguale a `y`. Tale insieme è il singoletto dell'elemento massimo di `A`, che è visualizzato con l'istruzione successiva.

Nell'esempio precedente abbiamo introdotto e manipolato un semplice esempio di insieme di interi, un oggetto i cui elementi sono tipi di dato semplici. Lo scopo del prossimo esempio è, data una rete dove i collegamenti tra i nodi hanno una direzione, determinare i nodi raggiungibili da ogni nodo. Possiamo modellizzare la rete con un grafo orientato.

**Definizione 1.2** *Un* ***grafo orientato*** $G$ *è una coppia* $(V, E)$ *dove:*

- $V$ *è un insieme finito di vertici;*
- $E \subseteq V \times V$ *insieme di archi.*

Un elemento di $E$, un arco, rappresenta quindi un collegamento della rete dove la prima componente rappresenta il nodo origine e la seconda componente rappresenta il nodo destinazione.
Un nodo $A$ raggiunge un nodo $B$ se esiste una sequenza di collegamenti che collega $A$ a $B$. I grafi orientati equivalgono alle relazioni binarie finite. La

raggiungibilità per ogni nodo è la chiusura transitiva della relazione $E$ sull'insieme $V$.

Di seguito è riportato il programma SETL che determina la chiusura transitiva `T` di una relazione `E`:

```
E:={["a","e"],["e","f"],["f","b"],["c","g"],["g","d"],
    ["d","g"]};
T:=E;
LOOP WHILE
    (EXISTS [A,B] IN T |
        (EXISTS [D,C] IN T |
            B=D AND [A,C] NOTIN T))
DO
  T:=T WITH [A,C];
END LOOP;
print("transitiveclosure("+E+")="+T);
```

Il frammento di programma produce il seguente output:

```
transitiveclosure({[a e] [c g] [d g] [e f] [f b] [g d]})=
{[a b] [a e] [a f] [c d] [c g] [d d] [d g] [e b] [e f]
[f b] [g d] [g g]}
```

Si noti che è sufficiente conoscere la definizione di relazione transitiva su un insieme per scrivere un algoritmo in SETL che determini la chiusura transitiva di una relazione data.
Il ciclo esterno itera fino a che esistono gli archi `[A,B]` e `[D,C]` tali che `B=D` e l'arco `[A,C]` non appartiene a `T`. A ogni iterazione `T` è un nuovo insieme con un elemento in più, l'arco `[A,C]`. Il ciclo esterno sfrutta il test quantificato `EXIST`, che permette di specificare la proprietà degli archi coinvolti; l'approccio è dichiarativo, infatti non interessa se implicitamente un test quantificato `EXIST` effettua un ciclo interno a un insieme.
L'insieme risultante `T` rappresenta la raggiungibilità per ogni nodo.

# Capitolo 2

# Insiemi in Java

Gli "insiemi standard" in Java sono definiti nelle API (Application Programming Interface) Java SE 6 di Sun Microsystems [7], con l'interfaccia `Set` e sono implementati nelle classi `HashSet` e `TreeSet` nel package `java.util`. Costituiscono una piccola parte della più ampia gerarchia che ha come radice l'interfaccia `Collection`.
In questo capitolo è data una breve descrizione della gerarchia, partendo dall'interfaccia `Collection` fino alla classe `HashSet` che è stata ampiamente utilizzata nel lavoro di tesi.

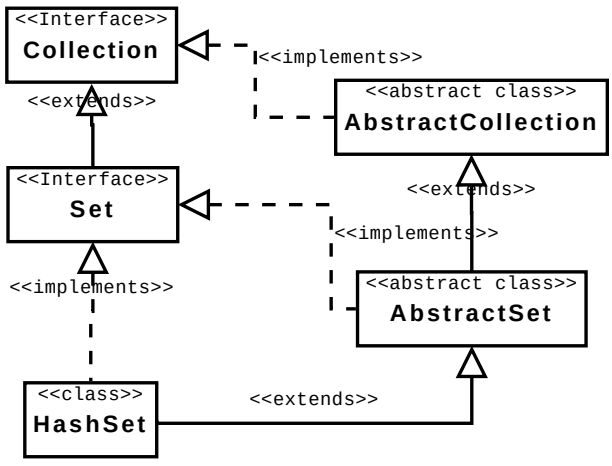


Figura 2.1: La gerarchia Collection per HashSet

## Notazioni

Una generica variabile di Java verrà indicata con una stringa che ha come primo simbolo una lettera latina minuscola. Una generica variabile può essere di tipo primitivo: `int`, `short`, `long`, `byte`, `char`, `boolean`, oppure istanza di una generica classe di Java. Una generica classe di Java è indicata con una stringa che ha come primo simbolo una lettera latina maiuscola.

## 2.1 L'interfaccia java.util.Collection

Una **collezione** rappresenta un *gruppo di oggetti detti elementi*.
Il concetto di collezione permette di astrarre diverse strutture dati: alcune permettono elementi duplicati, altre no, alcune sono ordinate, altre non lo sono e alcune hanno restrizioni sugli elementi o sul tipo di elementi che possono contenere.
Java non fornisce un'implementazione diretta di questa interfaccia; fornisce invece implementazioni di più specifiche interfacce, come `Set`. L'interfaccia `Collection` è tipicamente usata quando si desidera manipolare collezioni con il massimo grado di generalità.
Le implementazioni di `Collection`, per convenzione, forniscono almeno due costruttori:

- un costruttore senza argomenti, che crea una collezione vuota;
- un costruttore di un argomento di tipo `Collection`, che crea una nuova collezione con gli stessi elementi della `Collection` specificata (permette all'utente di copiare una qualsiasi collezione, producendo una collezione equivalente dell'implementazione voluta).

Non vi è modo di forzare questa convenzione (le interfacce non possono contenere costruttori), ma è rispettata da tutte le implementazioni di `Collection` nel package `java.util`.
L'interfaccia `Collection` definisce il contratto per i metodi:

- `boolean add(E e)`

  Assicura che la collezione oggetto di invocazione contenga l'elemento specificato (opzionale).

- `boolean addAll(Collection c)`

  Aggiunge tutti gli elementi della collezione specificata alla collezione oggetto di invocazione (opzionale).

- `void clear()`

  Rimuove tutti gli elementi dalla collezione oggetto d'invocazione collezione (opzionale).

- `boolean contains(Object o)`

  Ritorna vero se la collezione oggetto d'invocazione contiene l'elemento specificato.

- `boolean containsAll(Collection c)`

  Ritorna vero se la collezione oggetto d'invocazione contiene tutti gli elementi della collezione specificata.

- `boolean equals(Object o)`

  Confronta l'oggetto specificato con la collezione oggetto di invocazione per equivalenza.

- `int hashCode()`

  Ritorna il valore del codice hash per la collezione oggetto di invocazione.

- `boolean isEmpty()`

  Ritorna vero se la collezione oggetto di invocazione non contiene elementi.

- `Iterator iterator()`

Ritorna un iteratore sugli elementi della collezione oggetto di invocazione.

- `boolean remove(Object o)`

  Rimuove una singola istanza dell'elemento specificato dalla collezione, se l'elemento è presente (opzionale).

- `boolean removeAll(Collection c)`

  Rimuove tutti gli elementi della collezione oggetto di invocazione che sono anche elementi della collezione speficata (opzionale).

- `boolean retainAll(Collection c)`

  Mantiene solo gli elementi della collezione oggetto di invocazione che sono contenuti nella collezione specificata (opzionale).

- `int size()`

  Ritorna il numero di elementi della collezione oggetto di invocazione.

- `Object[] toArray()`

  Ritorna un array contenente tutti gli elementi della collezione oggetto di invocazione.

- `<T> T[] toArray(T[] a)`

  Ritorna un array contenente tutti gli elementi della collezione oggetto di invocazione; il tipo dell'array ritornato è quello dell'array specificato.

Alcuni di questi metodi definiti come opzionali possono non essere implementati e restituire eccezioni per le collezioni che non li supportano; sono tutti implementati invece per un oggetto di tipo insieme.

## 2.2 La classe astratta java.util.AbstractCollection

Questa classe fornisce un'*implementazione scheletrica dell'interfaccia* `Collection`, per minimizzare lo sforzo richiesto in una sua implementazione.

Sono dichiarati `abstract` i metodi `iterator` e `size`.
Per realizzare una collezione non modificabile è sufficiente estendere questa classe e implementare i metodi `iterator` e `size`; inoltre l'iteratore ritornato da `iterator` deve implementare i suoi metodi `next` e `hasNext` per permettere l'iterazione sugli elementi.
Per realizzare una collezione modificabile, va riscritto il metodo `add` che lancia un'eccezione; inoltre l'iteratore ritornato da `iterator` deve implementare i suoi metodi `next`, `hasNext` e `remove`.

## 2.3 L'interfaccia java.util.Set

Un `Set` è una `Collection` che *non contiene alcun elemento duplicato.*
Formalmente sono soddisfatte le seguenti proprietà:
Sia $A$ un `Set`

$$(\forall X, Y \in A) X \neq Y \longrightarrow \neg\exists(X, Y) : X.\texttt{equals}(Y)$$

$A$ non contiene alcuna coppia di elementi $X$ e $Y$ tali che $X.\texttt{equals}(Y)$

$$!!\exists X \in A : (X.\texttt{equals}(\texttt{null}))$$

$A$ ha al massimo un elemento nullo.
Soddisfacendo tali proprietà e visto che in una `Collection` gli elementi non hanno ordine, l'interfaccia `Set` *modella il concetto intuitivo di insieme.*
Tali proprietà impongono che, per un oggetto di tipo `Set`:

- ogni costruttore nella sua implementazione crei un insieme senza duplicati;
- l'inserimento di un elemento, con la `add`, avvenga se l'elemento non è già presente.

Occorre prestare attenzione all'inserimento di oggetti modificabili: il comportamento non è definito quando elementi dell'insieme sono modificabili e le modifiche influiscono sulla comparazione con `equals`.

## 2.4 La classe astratta java.util.AbstractSet

Questa classe fornisce un'*implementazione scheletrica dell'interfaccia* `Set`, per minimizzare lo sforzo richiesto in una sua implementazione.
Il processo di implementazione di una classe insieme che estende questa classe è simile a quello di implementazione di una collezione che estende `AbstractCollection`, in più devono essere rispettati i vincoli imposti dall'interfaccia `Set`.
In particolare `AbstractSet` sovrascrive il metodo `removeAll` di `Abstract Collection` e aggiunge l'implementazione dei metodi `equals` e `hashCode` .

## 2.5 La classe java.util.HashSet

La classe `HashSet` *implementa l'interfaccia* `Set`.
Questa classe non dá garanzie sull'ordine degli elementi e sulla loro iterazione, in particolare non garantisce che l'ordine rimanga costante nel tempo; permette come elemento `null`.
Per memorizzare gli oggetti `HashSet` utilizza le **tabelle hash** con al suo interno un oggetto di tipo `HashMap`.
Le tabelle hash [8] sono una generalizzazione del più semplice concetto di array ordinario; in un array l'indirizzamento diretto permette di esaminare una posizione arbitraria in tempo costante. Con il metodo hash a ogni elemento è associata una chiave $k$ ottenuta dall'universo delle chiavi $U = \{0, 1, ..., n-1\}$, si assume che nessuna coppia di elementi abbia la stessa chiave. L'elemento sarà memorizzato in un array $T$ di dimensioni inferiore alla cardinalità di $U$, in posizione $h(k)$ dove $h$ è una funzione detta funzione hash. Poichè la dimensione di $T$ è inferiore alla cardinalità di $U$ esistono almeno due chiavi con lo stesso valore hash. Tale fenomeno è chiamato **collisione**.
La strategia di risoluzione di collisioni utilizzata da `HashMap` è la strategia di concatenazione dove $T$ è un array di puntatori a liste, e gli elementi che collidono sono memorizzati nella stessa lista. Per minimizzare il numero di collisioni, la funzione hash deve distribuire le chiavi in $T$ con distribuzione

uniforme.
Per il calcolo del codice hash degli oggetti da memorizzare, `HashMap` sfrutta il metodo `hashCode` definito nella classe `Object`, e quindi invocabile su qualunque oggetto Java. Il metodo `hashCode` ritorna il codice hash di tipo `int` dell'oggetto su cui è stato invocato. Il metodo deve essere riscritto per ogni classe che riscrive il metodo `equals` e devono essere soddisfatte le proprietà:

- *consistenza durante l'esecuzione*: ogni volta che `hashCode` è richiamato sullo stesso `Object` durante l'esecuzione di un'applicazione Java, se l'oggetto non è stato modificato, `hashCode` deve restituire lo stesso intero; non è invece richiesta la consistenza tra l'esecuzione di una applicazione e un'altra esecuzione della stessa applicazione;

- *relazione con* `equals`: `Object` equivalenti con metodo `equals` devono avere lo stesso `hashCode` finché sono equivalenti; non è invece richiesto che oggetti non equivalenti con `equals` producano `hashCode` distinti.

Il numero intero restituito dal metodo `hashCode` viene quindi trasformato eseguendo un'operazione di modulo rispetto alla lunghezza dell'array.
In `HashMap` la dimensione dell'array $T$ è detta "capacity" (capacità). Un altro parametro in `HashMap` è il "load factor"(fattore di caricamento, in `HashMap` di tipo `float`, compreso tra 0 e 1), una misura di quanto piena dev'essere una tabella hash prima che la sua dimensione sia incrementata automaticamente. Quando il numero di elementi è maggiore del prodotto della capacità per il fattore di caricamento le strutture interne della `HashMap` sono ricostruite con il doppio della capacità (tale operazione è detta "rehash").
Il valore di default per il fattore di caricamento (0.75) offre un buon compromesso tra i costi in spazio e tempo; valori maggiori diminuiscono lo spazio non utilizzabile, ma aumentano il costo delle singole operazioni.
Quando si imposta la capacità iniziale, per minimizzare le operazioni di "rehash", vanno presi in considerazione il numero di elementi attesi e il fattore di caricamento: se la capacità iniziale è maggiore del massimo numero di

elementi diviso per il fattore di caricamento, non avverranno operazioni di "rehash".
In `HashSet` le operazioni `add`, `remove` e `contains` hanno un costo in tempo costante nel caso medio, nel caso peggiore costo lineare nel numero di elementi se tutti gli elementi collidono. L'iterazione invece richiede tempo proporzionale al numero di elementi dell'istanza di `HashSet` più la capacità dell'oggetto `HashMap` al suo interno, quindi, è importante non impostare la capacità uniziale troppo alta (o il fattore di caricamento troppo basso) se interessano prestazioni in iterazione.
I costruttori per la classe `HashSet` sono:

- `HashSet()`

  Costruisce un insieme vuoto; l'oggetto di classe `HashMap` ha valori di default per capacità iniziale(16) e fattore di caricamento(0.75).

- `HashSet(Collection c)`

  Costruisce un insieme contenente gli elementi della collezione specificata.

- `HashSet(int initialCapacity)`

  Costruisce un insieme vuoto; l'oggetto di classe `HashMap` ha il valore di capacità iniziale specificato e fattore di caricamento di default(0.75).

- `HashSet(int initialCapacity, float loadFactor)`

  Costruisce un insieme vuoto; l'oggetto di classe `HashMap` ha il valore di capacità iniziale specificato e fattore di caricamento specificato.

La classe `HashSet` inoltre implementa la classe `Cloneable`, fornisce l'implementazione del metodo `Object clone()` che crea e ritorna una copia superficiale dell'insieme dove gli elementi non sono clonati.

# Capitolo 3

# JSetL e principali strutture dati

I linguaggi di programmazione più diffusi, come C++ e Java, portano a pensare che oggi ci sia un solo modo di programmare, la programmazione ad oggetti (OO, Object Oriented), che prende il posto della più vecchia programmazione procedurale, come in C e Pascal.
Programmazione ad oggetti e programmazione procedurale hanno una importante caratteristica in comune: un programma consiste in **istruzioni** poste in sequenza o raggruppate con strutture di controllo che manipolano *dati* solitamente referenziati tramite **variabili**. Linguaggi di programmazione di questo "tipo" descrivono *come* un problema deve essere risolto e fanno parte del *paradigma di programmazione imperativo*.
Esistono però altre tecniche che si dimostrano convenientemente applicabili in determinati contesti. La *programmazione logica* (LP, Logic Programming) è un diverso paradigma di programmazione che adotta la logica del primo ordine per rappresentare e per elaborare l'informazione. Questo diverso modo di rappresentare un problema ne consente una comprensione **dichiarativa**, cioè si pone l'attenzione su *cosa* un programma deve risolvere, senza specificare come il problema verrà risolto.
Il compito di trovare il modo migliore per risolvere il problema è invece a

carico dell'interprete del linguaggio. Linguaggi logici **general-purpose**, come Prolog, sono strumenti validi per lo sviluppo di applicazioni complesse, soprattutto nel campo dell'intelligenza artificiale.
Approcci simili sono utilizzati per sistemi **special-purpose**, alcuni di uso comune come SQL. Con SQL infatti il programmatore si limita a specificare la forma che dovrà avere il risultato dell'operazione, ad esempio quali tabelle considerare, quali condizioni applicare, come raggruppare le righe, come ordinare il risultato.
La *programmazione logica a vincoli* (CLP, Constraint Logic Programming) estende la programmazione logica con un meccanismo di *risoluzione di vincoli* (relazioni su opportuni domini sia simbolici che numerici) al fine di superarne le principali limitazioni quali l'unificazione sintattica e l'inefficienza di esplorazione dello spazio di soluzioni. Nella programmazione logica a vincoli è necessario definire **quali vincoli** si trattano e su **quale dominio** di computazione. La collezione di vincoli durante il corso della computazione è detta **constraint store**. Sul constraint store sono applicabili due operazioni: la verifica di consistenza e la propagazione di vincoli. Queste operazioni dipendono e sono definite dal dominio di computazione e dai vincoli considerati. È detta **risposta calcolata** il contenuto del constraint store, ristretto alle variabili che interessano, al termine di una computazione di successo.
Recentemente si è tentato di far *coesistere* il paradigma di programmazione logica con quello di programmazione tradizionale orientato agli oggetti in diversi modi: definendo nuovi linguaggi, estendendone di già esistenti oppure inserendo nel linguaggio, detto **linguaggio ospite**, una nuova libreria scritta nello stesso linguaggio. Questa ultima soluzione è quella più adottata.

JSetL [5] è una libreria Java che *combina la programmazione object-oriented di* Java *con i concetti fondamentali della programmazione* CLP, come variabili logiche, liste anche parzialmente specificate, unificazione, risoluzione di vincoli, non-determinismo. La libreria fornisce anche insiemi e risoluzione di vincoli su insiemi come unione, intersezione, inclusione, e altri, come in

CLP($\mathcal{SET}$) [2]. La *risoluzione di vincoli su insiemi* in JSetL avviene in modo **non-deterministico**, con **punti di scelta** e **backtracking**. Ne segue che anche la soluzione di semplici problemi può comportare una complessità computazionale elevata, ma l' efficienza non è l'obiettivo primario in JSetL. JSetL è stato sviluppato presso il Dipartimento di Matematica dell' Università di Parma. Si tratta di un package completamente scritto in codice Java, importabile con la sola istruzione

```
import JSetL.*;
```

La libreria è un software libero, sotto i termini della licenza GNU. La versione corrente è JSetL 1.3, ma quella utilizzata nel lavoro di tesi è allineata all'ultima versione disponibile, versione comprensiva delle modifiche proposte nelle tesi di Delia Di Giorgio [1], la creazione dell'interfaccia `MutableLSet`, Roberto Amadini [9], modifiche che coinvolgono il trattamento del vincolo di cardinalità insiemistica, e Daniele Pandini [10], modifiche che introducono un risolutore di vincoli su domini finiti all'interno del risolutore di vincoli insiemistici fornito da JSetL.
In questo capitolo sono elencate le principali strutture dati in JSetL, con particolare attenzione agli insiemi logici, che saranno utilizzati nel seguito.

## 3.1 Variabili logiche

**Definizione 3.1** *Una **variabile logica** è un'istanza della classe* `Lvar`*, creata dalla dichiarazione*

```
Lvar nomeVar = new Lvar(NomeVarExt, ValoreVar);
```

dove `nomeVar` è il nome dell'oggetto Java, `NomeVarExt` un nome esterno (parametro opzionale), `ValoreVar` è un valore (parametro opzionale) di inizializzazione della variabile stessa.

**Definizione 3.2** *Una **variabile logica** che non ha un valore associato con*

*essa è detta **non inizializzata** (o incognita). Altrimenti la variabile è detta **inizializzata**.*

Il valore di una variabile logica può essere specificato alla creazione o come il risultato di elaborazione di vincoli che la coinvolgono, in particolare, vincoli di uguaglianza. Oltre ai vincoli, la classe `Lvar` fornisce metodi che permettono di leggere e scrivere il valore della variabile logica, di conoscere se la variabile inizializzata o no, di ottenere il suo nome esterno, e cosi via.

## 3.2 Liste logiche

**Definizione 3.3** *Una **lista logica** in JSetL è un'istanza della classe* `Lst`*, creata dalla dichiarazione*

```
Lst nomeLst = new Lst(NomeLstExt, ValoreLstElem);
```

dove `nomeLst` è il nome dell'oggetto Java, `NomeLstExt` è un nome esterno (paramentro opzionale), `ValoreLstElem` 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 liste in JSetL possono essere inizializzate o non inizializzate e gli elementi possono essere di qualsiasi tipo Java. Il valore di una lista può essere specificato elencando i suoi elementi in un array o, implicitamente, fornendo i limiti di un intervallo di numeri interi, o attraverso un'altra lista.

## 3.3 Insiemi logici

**Definizione 3.4** *Un **insieme logico** in JSetL è un'istanza della classe* `LSet`*, creata dalla dichiarazione*

```
LSet nomeSet = new ConcreteLSet(NomeSetExt, ValoreSetElem);
```

dove `nomeSet` è il nome dell'oggetto Java, `NomeSetExt` è un nome esterno

(paramentro opzionale), `ValoreSetElem` è usato per specificare gli elementi dell'insieme (parametro opzionale) e può essere un array di elementi di qualsiasi tipo, o i limiti `l` e `u` di tipo primitivo `int` di un intervallo [`l`,`u`] di numeri interi. Se `ValoreSetElem` non è specificato è 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 nomeSet = new ConcreteLSet(NomeSetExt,arr);
LSet nomeSet = new ConcreteLSet(NomeSetExt,l,u);
LSet nomeSet = ConcreteLSet.empty;
```

Nella prima dichiarazione `nomeSet` è un insieme che ha per elementi gli elementi di un array di tipo primitivo.
Nella seconda dichiarazione `nomeSet` è un insieme che ha per elementi i numeri interi dell'intervallo [`l`,`u`].
Nella terza dichiarazione `nomeSet` è l'insieme vuoto.

**Definizione 3.5** *Un insieme (o lista) che contiene alcuni elementi che non sono inizializzati è detto* ***insieme (o lista) parzialmente specificato****.*

**Definizione 3.6** *Gli elementi di un insieme (o lista) che sono essi stessi insiemi (o lista) sono detti* ***insiemi (o liste) annidati****.*

**Definizione 3.7** *Un insieme logico (o lista) che ha tutti gli elementi specificati è detto* ***insieme logico (o lista) ground***

Variabili logiche, insiemi logici e liste logiche sono per definizione immutabili. Insiemi e liste logici possono essere manipolati attraverso appositi metodi di inserimento ed estrazione che non modificano l'oggetto di invocazione ma ne restituiscono uno nuovo. Tali metodi sono costruttori di insiemi o liste rispettivamente.

**Definizione 3.8** *Un'espressione che costruisce e restituisce un nuovo insieme (o lista) è detta* ***costruttore di insiemi (o liste).***

Siano `S` un insieme JSetL, `e` un oggetto JSetL, `O` un array di oggetti, `NomeSetExt` una `java.lang.String`. Sono costruttori di insiemi logici espressioni del tipo:

```
S.ins(e)
S.insAll(O)
ConcreteLSet.mkset(NomeSetExt,n)
```

La prima espressione restituisce l'insieme ottenuto dall'unione di `S` con il singoletto dell'elemento `e`.
La seconda espressione restituisce l'insieme ottenuto dall'unione di `S` con l'insieme costituito dagli oggetti dell'array `O`.
La terza espressione restituisce un nuovo insieme costituito da `n` variabili logiche non inizializzate di nome esterno `NomeSetExt` (parametro opzionale).

Analogamente per le liste, siano `L` una lista `Lst`, `e` un oggetto JSetL, `O` un array di oggetti, `X` una `Lvar` non inizializzata e `n` di tipo `int`. Sono costruttori di liste logiche espressioni del tipo:

```
L.ins1(e);
L.insn(e);
L.ins1All(O);
L.insnAll(O);
L.ext1(X);
L.extn(X);
Lst.mkLst(NomeSetExt,n)
```

I costruttori con `ins` hanno significato analogo ai costruttori di insiemi, `ext` indica la rimozione di un elemento che sarà assegnato a `X`, "`1`" e "`n`" in-

dicano che l'operazione è effettuata rispettivamente sulla testa o coda della lista.
L'ultima espressione restituisce una lista contenente `n` variabili logiche non inizializzate di nome esterno `NomeSetExt` (parametro opzionale).

**Definizione 3.9** *Un **insieme (o lista)** è detto **illimitato** se contiene un certo numero di elementi* e1,e2,...,en *e un resto* R *rappresentato da un insieme (o lista) non inizializzato. La notazione astratta è la seguente* {e1,e2,...,en|R} *(o* [e1,e2,...,en|R]*).*

Se `S` è un insieme non inizializzato, con i costruttori di insiemi su `S` è possibile definire insiemi illimitati:

```
S.ins(e);
S.insAll(O);
```

La prima espressione crea un insieme illimitato che ha come resto l'insieme `S` e come elemento `e`.
La seconda espressione crea un insieme illimitato che ha come resto l'insieme `S` e per elementi gli oggetti dell'array `O`.

Seguono alcuni esempi di dichiarazioni di insiemi.

**Esempio 3.1**

```
LSet ls0 = ConcreteLSet.empty;
LSet ls1 = ls0.ins(1);
Lvar x = new Lvar();
LSet ls2 = ls1.ins(x);
```

Crea un insieme vuoto `ls0` e a partire da questo un insieme di un elemento `ls1` ground e da quest'ultimo un insieme `ls2` parzialmente specificato, {1, `x`}.

**Esempio 3.2**

```
int[] elems = 1, 2, 3;
LSet ls3 = ConcreteLSet.empty.insAll(elems);
```

Crea un insieme `ls3` ground, {1, 2, 3}, di elementi gli elementi dell'array `elems` a partire dall'insieme vuoto.

**Esempio 3.3**

```
LSet ls4 = new ConcreteLSet(``interv'',1,5);
```

Crea un insieme `ls4` ground, {1, 2, 3, 4, 5}, di nome esterno `interv` di elementi gli interi dell'intervallo [1,5].

**Esempio 3.4**

```
LSet ls5 = new ConcreteLSet();
LSet ls6 = ls5.ins(1);
```

Crea un insieme non inizializzato `ls5` e a partire da questo tramite assegnamento `ls6` è un insieme illimitato, { 1 | `ls5` }.

# Capitolo 4

# Il sistema integrato: visione utente

Obiettivo del lavoro di tesi è di completare e di rendere più **efficiente** e **usabile** l'astrazione di insieme, realizzata dalla classe `MutableLSet`, creata nel lavoro di tesi di Delia Di Giorgio [1]. Dal punto di vista utente questa astrazione permette di operare sugli insiemi sia con metodi forniti dagli "insiemi standard" Java sia con quelli degli "insiemi logici" JSetL. L'interfaccia `MutableLSet` è chiamata così in quanto un oggetto di questo tipo è **mutabile**, cioè attraverso una chiamata a un metodo l'oggetto di invocazione può essere modificato. Anche un insieme `java.util.Set` può essere modificato; un insieme `LSet` invece è un oggetto immutabile. Una corretta documentazione utente, non presente in [1], chiarisce le differenze d'uso tra insiemi `LSet`, insiemi `java.util.Set`, e insiemi `MutableLSet`.

## 4.1 Insiemi logici mutabili

**Definizione 4.1** *Un **insieme logico mutabile** in JSetL è un'istanza della classe* `MutableLSet` *creata dalla dichiarazione:*

```
MutableLSet nomeSet = new ConcreteMutableLSet(NomeSetExt,
    ValoreSetElem);
```

dove `nomeSet` è il nome dell'oggetto Java, `NomeSetExt` è un nome esterno (paramentro opzionale), `ValoreSetElem` è usato per specificare gli elementi dell'insieme (parametro opzionale) e può essere una `java.util.Collection`, un array di tipo primitivo `int`, o i limiti `l` e `u` di tipo `int` di un intervallo [`l`,`u`] di numeri interi. Se `ValoreSetElem` non è specificato è costruito un insieme non inizializzato. Come importante conseguenza, contrariamente alle convenzioni degli oggetti di tipo `Collection`, ma come avviene per gli insiemi logici `LSet`, il costruttore senza parametri crea un insieme non inizializzato.
Le dichiarazioni hanno quindi la forma, con `intarr` array di tipo primitivo `int`, `C` di tipo `Collection`, `l` e `u` di tipo primitivo `int`, `n` di tipo primitivo `int`:

```
MutableLSet nomeSet = new ConcreteMutableLSet(NomeSetExt);
MutableLSet nomeSet = new ConcreteMutableLSet(NomeSetExt,C);
MutableLSet nomeSet = new ConcreteMutableLSet(NomeSetExt,
    intarr);
MutableLSet nomeSet = new ConcreteMutableLSet(NomeSetExt,l,u);
```

Nella prima dichiarazione `nomeSet` è un insieme non inizializzato.
Nella prima dichiarazione `nomeSet` è un insieme di elementi gli elementi della collezione specificata. Se la collezione è vuota `nomeSet` è un insieme vuoto.
Nella seconda dichiarazione `nomeSet` è un insieme di elementi gli elementi di un array di tipo primitivo `int`.
Nella terza dichiarazione `nomeSet` è un insieme di elementi gli interi dell'intervallo [`l`,`u`].
Per tutti i costruttori la documentazione in stile Javadoc è riportata in appendice A.1.
Come per gli insiemi logici gli insiemi logici mutabili possono essere inizializzati o non inizializzati, limitati o illimitati, parzialmente specificati, o ground.
Oltre ai costruttori della classe, `MutableLSet` fornisce alcuni altri metodi che permettono di costruire un insieme.

A differenza di un insieme logico `LSet` non vi è una costante che denota l'insieme vuoto; l'insieme vuoto è un insieme mutabile privo di elementi, ottenibile con il costruttore di insiemi:

```
ConcreteMutableLSet.empty()
```

Siano `S` un insieme JSetL, `e` un oggetto JSetL, `O` un array di oggetti, `NomeSetExt` una `java.lang.String`. Sono costruttori di insiemi logici mutabili espressioni del tipo:

```
S.ins(e)
S.insAll(O)
ConcreteMutableLSet.mkset(NomeSetExt,n)
```

La prima espressione restituisce l'insieme ottenuto dall'unione di `S` con il singoletto dell'elemento `e`.
La seconda espressione restituisce l'insieme ottenuto dall'unione di `S` con l'insieme costituito dagli oggetti dell'array `O`.
La terza espressione restituisce un nuovo insieme costituito da `n` variabili logiche non inizializzate di nome esterno `NomeSetExt` (parametro opzionale).

## 4.2 Operazioni su MutableLSet

I metodi di `MutableLSet` visibili all'utente sono costituiti da metodi ereditati da `java.util.Set`, metodi ereditati da `LSet` e nuovi metodi aggiunti per rendere più flessibile l'utilizzo.
Dal punto di vista dell'utente questi metodi si possono dividere in metodi di utilità e metodi di generazione di vincoli.

## 4.3 Metodi di utilità

I metodi di utilità sono metodi che restituiscono informazioni sull'insieme o che eseguono operazioni su di esso.

Per i metodi di utilità la documentazione in stile Javadoc è riportata in appendice A.2.
Nel seguito è riportato tramite semplici esempi il funzionamento del metodo `output` applicato ad insiemi ottenuti in vario modo (con costruttori e metodi `add` e `ins`).

**Insiemi non inizializzati**

`MS0` e `S0` sono insiemi non inizializzati, rispettivamente di tipo `MutableLSet` e `LSet`.

```
...
MutableLSet MS0=new ConcreteMutableLSet();
LSet S0=new ConcreteLSet();
MS0.output();
S0.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_0 = unknown
LSet_2 = unknown
```

**Insiemi vuoti**

`MS0`, `S0` e `MS1` sono insiemi vuoti, in particolare `MS1` è costruito passando al costruttore di parametro `java.util.Collection` una collezione vuota.

```
...
MS0=ConcreteMutableLSet.empty();
S0=ConcreteLSet.empty;
MutableLSet MS1=new ConcreteMutableLSet(new Vector());
MS0.output();
S0.output();
MS1.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_1 = {}
emptySet = {}
MutableLSet_2 = {}
```

Si noti che `S0` ha nome `emptySet`; l'insieme vuoto `ConcreteLSet` è unico.

**Insiemi ground dati per intervallo**

`MS0` e `S0` sono insiemi ground di elementi i numeri interi in un intervallo specificato, costruiti con il costruttore di parametri due interi. `MS1`, `MS2` e `S2` sono costruiti partendo dall'insieme vuoto e inserendo elemento per elemento, in particolare `MS1` è costruito con metodo `add`, mentre `MS2` e `S2` sono costruiti con il costruttore di insiemi `ins` (la `add` è possibile solo per `MutableLSet`).

```
...
int n=10;
MS0=new ConcreteMutableLSet(0,n-1);
S0=new ConcreteLSet(0,n-1);
MutableLSet MS2=ConcreteMutableLSet.empty();
LSet S2=ConcreteLSet.empty;
for (int i=0;i<n;i++) {
    MS1.add(i);
    MS2=MS2.ins(i);
    S2=S2.ins(i);
}
MS0.output();
S0.output();
MS1.output();
MS2.output();
S2.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_3 = {0,1,2,3,4,5,6,7,8,9}
LSet_3 = {0,1,2,3,4,5,6,7,8,9}
MutableLSet_4 = {0,1,2,3,4,5,6,7,8,9}
MutableLSet_16 = {0,1,2,3,4,5,6,7,8,9}
LSet_13 = {9,8,7,6,5,4,3,2,1,0}
```

Si noti che gli elementi di `S2` sono visualizzati con un diverso ordine rispetto agli elementi di `MS0`, `S0`, `MS1` e `MS2`, ma gli insiemi sono equivalenti.

**Insiemi annidati costruiti con inserimento**

`MS1`, `MS2` e `S2` sono insiemi annidati, in particolare `MS1` è costruito con metodo `add`, mentre `MS2` e `S2` sono costruiti con costruttori di insiemi `ins`.

```
...
MS1.add(MS0);
MS2=MS2.ins(MS0);
S2=S2.ins(S0);
MS1.output();
MS2.output();
S2.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_20 = {0,1,2,3,4,5,6,{0,1,2,3,4,5,6,7,8,9,10},7,8,
    9,10}
MutableLSet_28 = {0,1,2,3,4,5,6,7,8,9,10,{0,1,2,3,4,5,6,7,8,
    9,10}}
LSet_19 = {{10,0,1,2,3,4,5,6,7,8,9},10,9,8,7,6,5,4,3,2,1,0}
```

**Insiemi illimitati costruiti con inserimento**

`MS1` e `S1` sono insiemi illimitati, costruiti a partire dagli insiemi non inizializzati con costruttori di insiemi `ins`.

```
...
MS1=new ConcreteMutableLSet();
LSet S1=new ConcreteLSet();
MS1=MS1.insAll(MS0);
Vector v=S0.toVector();
Iterator i=v.iterator();
while (i.hasNext()) S1=S1.ins(i.next());
MS1.output();
S1.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_30 = {0,1,2,3,4,5,6,7,8,9,10|MutableLSet_29}
LSet_33 = {9,8,7,6,5,4,3,2,1,0,10|LSet_21}
```

**Insiemi parzialmente specificati**

`MS0`, `S0`, `MS1`, `MS2`, `S2` sono insiemi parzialmente specificati, costruiti partendo da insiemi ground con costruttori di insiemi `ins`.

```
...
Lvar X=new Lvar();
MS0=MS0.ins(X);
S0=S0.ins(X);
MS1=new ConcreteMutableLSet(0,10);
MS1=MS1.ins(X);
MS2=MS2.ins(X);
S2=S2.ins(X);
MS0.output();
S0.output();
MS1.output();
MS2.output();
S2.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_18 = {_Lvar_0,0,1,2,3,4,5,6,7,8,9}
LSet_15 = {_Lvar_0,0,1,2,3,4,5,6,7,8,9}
MutableLSet_20 = {_Lvar_0,0,1,2,3,4,5,6,7,8,9}
MutableLSet_22 = {_Lvar_0,0,1,2,3,4,5,6,7,8,9}
LSet_18 = {_Lvar_0,9,8,7,6,5,4,3,2,1,0}
```

L'esempio sarà ripreso in un secondo momento dopo l'introduzione dei metodi di generazione di vincoli.

## 4.4 Metodi di generazione di vincoli

I vincoli in JSetL specificano le proprietà di variabili logiche, liste logiche e insiemi logici; il dominio dei vincoli è il dominio definito in $\mathcal{SET}$[2], esteso con alcuni vincoli per il confronto su interi.

**Definizione 4.2** *Un **vincolo atomico** in JSetL è un'istanza della classe* `Constraint` *creata con un'espressione dalla forma:*

```
e1.op(e2)
e1.op(e2,e3)
```

dove `op` è un metodo predefinito di generazione di vincoli atomici e `e1`, `e2`, `e3` sono espressioni che dipendono da `op`.
Nel caso `e1` di tipo `MutableLSet` oggetto di invocazione, `op` è un metodo di generazione di vincoli atomici cioè di tipo di ritorno `Constraint`, `e1` e `e2` sono nomi di oggetti Java e il loro tipo dipende dal metodo `op`.

**Definizione 4.3** *Una **congiunzione di vincoli** in JSetL è un'istanza della classe* `ConstraintConjunction` *creata con un'espressione dalla forma:*

```
c1.and(c2)...  .and(cn)
e1.op2(e2)
```

```
e1.op2(e2,e3)
```

dove `c1`, `c2`, ...,`cn` sono vincoli atomici, `op2` è un metodo predefinito di generazione di congiunzioni di vincoli e `e1`, `e2`, `e3` sono espressioni che dipendono da `op2`.
La semantica della prima espressione è la congiunzione logica della semantica dei vincoli atomici `c1`, `c2`, ...,`cn`. Nella seconda e terza espressione, nel caso `e1` di tipo `MutableLSet` oggetto di invocazione, `op2` è un metodo di generazione di congiunzioni di vincoli cioè di tipo di ritorno `ConstraintConjunction`, `e1` e `e2` sono nomi di oggetti Java e il loro tipo dipende dal metodo `op2`.

**Definizione 4.4** *Un **vincolo** in JSetL è un vincolo atomico o una congiunzione di vincoli.*

I vincoli sono creati con metodi di generazione di vincoli e passati a un risolutore di vincoli. Per i metodi di generazione di vincoli della classe `MutableLSet` la documentazione in stile Javadoc è riportata in appendice A.3.

**Definizione 4.5** *Un **risolutore di vincoli** (Constraint Solver) in JSetL è un'istanza della classe* `SolverClass` *creata dalla dichiarazione:*

```
SolverClass nomeSolver = new SolverClass();
```

dove `nomeSolver` è il nome dell'oggetto Java.
Quando si passa un vincolo a un risolutore d vincoli, questo è aggiunto al **Constraint Store** (CS), *la collezione di vincoli attivi.* L'operazione che permette l'inserimento di un vincolo nel CS è il metodo `add` della classe `SolverClass`, l'inserimento ha la forma:

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

dove `S` è il nome dell'oggetto Java di tipo `SolverClass` e `C` denota un vincolo. La semantica della collezione di vincoli nel CS è la congiunzione logica dei vincoli che contiene, l'ordine di inserimento è irrilevante.
Il soddisfacimento dei vincoli è verificato con il metodo `solve` della classe `SolverClass`:

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

dove `S` è il nome dell'oggetto Java di tipo `SolverClass`.
Se i vincoli non possono essere soddisfatti il metodo genera un'eccezione `Failure`. I vincoli possono contenere variabili logiche non inizializzate e permettere computazioni con dati parzialmente specificati. Può essere restituito l'insieme di tutte le soluzioni nei seguenti modi, con `S` nome dell'oggetto Java di tipo `SolverClass` e `x` una variabile logica non inizializzata:

```
S.setof(x);
S.mutableSetof(x);
```

La prima espressione crea un insieme `LSet`; la seconda espressione crea un insieme `MutableLSet`.
La risoluzione di vincoli in JSetL è non deterministica, usa punti di scelta e backtracking; una trattazione specifica è rimandata all'articolo *JSetL:a Java library for supporting declarative programming in Java* [6].

**Insiemi che diventano ground**

`MS0`, `S0`, `MS1`, `MS1`, `S2` sono insiemi ground ottenuti dagli insiemi parzialmente specificati in 4.1 con risoluzione del vincolo di uguaglianza della variabile logica non inizializzata con l'intero specificato.

```
...
SolverClass solver=new SolverClass();
solver.add(X.eq(10));
solver.solve();
MS0.output();
```

```
S0.output();
MS1.output();
MS2.output();
S2.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_18 = {10,0,1,2,3,4,5,6,7,8,9}
LSet_15 = {10,0,1,2,3,4,5,6,7,8,9}
MutableLSet_20 = {10,0,1,2,3,4,5,6,7,8,9}
MutableLSet_22 = {10,0,1,2,3,4,5,6,7,8,9}
LSet_18 = {10,9,8,7,6,5,4,3,2,1,0}
```

Si noti che il metodo `output` visualizza il valore della variabile logica che è un oggetto dell'insieme.

**Insiemi ground**

`MS0`, `MS1`, `MS2` sono insiemi ground.

```
...
MS0.add(1);
MS1.add(1);
MS2.add(1);
MS0.output();
MS1.output();
MS2.output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_18 = {0,1,2,3,4,5,6,7,8,9,10}
MutableLSet_20 = {0,1,2,3,4,5,6,7,8,9,10}
MutableLSet_22 = {0,1,2,3,4,5,6,7,8,9,10}
```

Si noti che l'inserimento del valore con metodo `add` apparentemente non modifica l'insieme; dal punto di vista implementativo, analizzato successivamente in 5.4, gli insiemi `MS0`, `MS1`, `MS2` hanno una diversa rappresentazione interna.

## 4.5 La documentazione utente

La documentazione utente per l'interfaccia `MutableLSet` è realizzata utilizzando lo strumento di generazione automatica di documentazione **Javadoc** di cui daremo una breve descrizione mostrando esempi relativi a JSetL.
Javadoc è uno dei programmi di utilità inclusi nel SDK[1] di Sun Microsystems [7]. Grazie ad esso è possibile generare in modo automatico la documentazione in formato HTML dai sorgenti dei programmi Java, a patto che vengano rispettate regole precise nella scrittura dei commenti stessi. Un commento al codice Java per poter essere interpretato da Javadoc deve essere racchiuso fra le sequenze di caratteri `/**` e `*/`. Di fatto è una forma particolare di commento multilinea, che sarà aggiunto alla documentazione dell'elemento che lo segue dal tool Javadoc. Il comando per generare la documentazione inerente a un file `nomeclasse.java` è:

```
javadoc nomeclasse.java
```

Javadoc è quindi un facile strumento per la generazione automatica della documentazione utente e utile per manutenzione, o riuso del codice; permette di creare con facilità documentazione dall'aspetto professionale, secondo le linee guida della documentazione che Sun fornisce per le API[2] che distribuisce. Un commento di documentazione Javadoc deve precedere una classe, un campo, un costruttore o la dichiarazione di un metodo; è composto da due parti, la descrizione dell'elemento seguita da opportuni tag.
Un esempio di documentazione in stile Javadoc nell'interfaccia `MutableLSet`:

[1] Software Development Kit

[2] Application Programming Interface: insieme di procedure disponibili al programmatore raggruppate in strumenti specifici per un determinato compito

```
/**
 * Aggiunge tutti gli elementi dell'array di interi
 * specificato a questo insieme se non sono già presenti
 * (eccezione se questo insieme non è ground).
 *
 * @param arr array di int da aggiungere
 * @return vero se questo insieme è modificato
 */
public boolean addAll(int[] arr){ ... }
```

Le prime tre righe di commento descrivono il metodo, `@param` e `@return` sono tag che indicano rispettivamente le informazioni sui parametri e sul valore di ritorno.
Di seguito un elenco dei tag più usati in Javadoc:

| tag | descrizione |
|---|---|
| `@author` | Indica il nome dello sviluppatore. |
| `@deprecated` | Indica che l'elemento potrà essere eliminato da una versione successiva del software. |
| `@throws` | Indica le eccezioni lanciate da un metodo. |
| `@link` | Crea un collegamento ipertestuale a documentazione locale o risorse esterne. |
| `@param` | Indica i parametri di un metodo, richiesto per ogni parametro. |
| `@return` | Indica il valore di ritorno di un metodo, non va usato se il metodo restituisce void. |
| `@see` | Indica un'associazione a un altro metodo o classe. |
| `@since` | Indica quando un metodo è stato aggiunto a una classe. |
| `@version` | Indica il numero di versione di una classe o un metodo. |

Con Javadoc si pone l'accento sulle condizioni al contorno, range per argomenti, o casi limite piuttosto che descrivere casi comuni, o mostrare esempi. I principi della scrittura dei commenti secondo le specifiche API sono:

- Le specifiche API sono definite nei commenti di documentazione del codice sorgente e in qualsiasi documento denotato come specifiche e raggiungibile da questo.
- Le specifiche API sono un contratto tra chiamante e implementazione.
- Quando non specificato, le affermazioni devono essere indipendenti dall'implementazione.
- Le affermazioni non devono commentare bug o comportamenti anomali.

Inoltre per la scrittura di una documentazione leggibile occorre seguire una serie di convenzioni:

- Usare la terza persona singolare per le affermazioni.
- Iniziare con un verbo le descrizioni dei metodi.
- Usare frasi anche non complete, per le descrizioni di classi, interfacce o campi, il verbo può non essere necessario.
- Usare "questo" per riferirsi all'oggetto di invocazione.
- Omettere le parentesi quando ci si riferisce a un metodo o a un costruttore, se questo ha più forme usare le parentesi e per l'argomento specificarne soltanto il tipo.
- Aggiungere una descrizione al nome dell'oggetto, in java i nomi degli oggetti danno informazioni sull'oggetto, non ha senso ripeterne solo il nome.
- Racchiudere le parole chiave tra i tag `<code>...</code>`; parole chiave possono essere nomi di package, classi, ...

- Non abusare del tag `@link`, tenere presente che un collegamento ipertestuale cattura l'attenzione del lettore.

Tutta la documentazione prodotta per la classe `MutableLSet` cerca di rispettare le linee guida API e le convenzioni per una documentazione leggibile. Come esempio è riportata la documentazione di due metodi:

```
/**
 * Costruisce il vincolo che impone che tutti gli elementi
 * di questo insieme siano diversi fra loro, implementato
 * con decomposizione binaria.
 *
 * @return Constraint vincolo costruito
 */
 public Constraint allDifferent(){ ... }

/**
 * Costruisce il vincolo che impone che tutti gli elementi
 * di questo insieme siano diversi fra loro, implementato
 * come vincolo globale.
 *
 * @return Constraint vincolo costruito
 */
 public Constraint allDistinct(){ ... }
```

I due metodi costruiscono lo stesso vincolo e differiscono soltanto per l'implementazione (una trattazione specifica si trova in [10]). Darne una breve descrizione è necessario e non viola le specifiche. Si noti che entrambe le descrizioni iniziano con il verbo "`Costruisce`", il soggetto è il metodo e sono nella terza persona singolare, per i valori di ritorno non sono usate frasi complete, è usato "`questo insieme`" per riferirsi all'insieme oggetto di invocazione e non è stato usato il tag `@link`.

# Capitolo 5

# Il sistema integrato: sviluppo

Uno degli obiettivi del lavoro di tesi è di rendere **efficiente** l'utilizzo degli insiemi definiti tramite la classe `MutableLSet`. Questa classe permette di operare sugli insiemi sia con metodi forniti da `java.util.Set` sia con quelli di JSetL e mantiene la compatibilità con le applicazioni scritte utilizzando le precedenti librerie.
Riguardo l'implementazione della classe `MutableLSet` la soluzione proposta in [1] simulava il comportamento dei metodi di `java.util.Set` con metodi di JSetL risultando così particolarmente inefficiente. In questo capitolo si descrivono le modifiche apportate alla libreria JSetL dall'ultima versione disponibile in [10].

## 5.1 Architettura

Perché l'interfaccia `MutableLSet` fornisca all'utente sia le funzionalità degli insiemi Java sia le funzionalità degli insiemi JSetL si è utilizzata l'ereditarietà dalle intefacce `LSet` (interfaccia che contiene tutti i metodi visibili all'utente per un insieme JSetL) e `java.util.Set`. `MutableLSet` è un'interfaccia e, a differenza delle classi in Java che possono ereditare al più da un'altra classe, può estendere le due interfacce. Tale relazione è di tipo is-a, cioè un `MutableLSet` è allo stesso tempo un `LSet` e un insieme Java e può

essere inserito in entrambi i contesti con minimi e documentati problemi di compatibilità.
La definizione dell'interfaccia `MutableLSet` è la seguente:

```
public interface MutableLSet extends java.util.Set, LSet{
  //NUOVI METODI
  ... }
```

`MutableLSet` introduce anche alcuni nuovi metodi: precisamente un metodo `add` che permette di inserire in un insieme un array di `int`, e un metodo `ins` che permette di inserire in un insieme gli elementi di una `java.util.Collection`.

Tramite l'interfaccia `MutableLSet` (e la sua implementazione `ConcreteMutableLSet`) è possibile creare oggetti che possano utilizzare sia i metodi di `java.util.Set` sia i metodi pubblici degli insiemi JSetL, nei modi ampiamente documentati in 4.1.
Inoltre è possibile creare oggetti che possano utilizzare esclusivamente i metodi di una delle due interfacce, ad esempio:

```
LSet s2 = new ConcreteMutableLSet();
java.util.Set s3 = new ConcreteMutableLSet();
```

Nella prima dichiarazione `s2` è un oggetto di tipo apparente `LSet` e tipo reale `ConcreteMutableLSet` e fornisce i medodi dell'interfaccia `LSet`.
Nella seconda dichiarazione `s3` è un oggetto di tipo apparente `java.util.Set` e tipo reale `ConcreteMutableLSet` e fornisce i medodi dell'interfaccia `java.util.Set`.

`LSet` definisce i metodi visibili all'utente, ma *l'implementazione di un* "insieme logico" *necessita di altri metodi*, usati da classi interne a JSetL, *che non devono essere visibili all'esterno* del package JSetL. Un'interfaccia Java permette esclusivamente di definire metodi con i modificatori `public`,

`static` e `final`; in `LSet` non è possibile definire metodi con visibilità interna al package. Per questo motivo è prevista l'interfaccia `LSetProtected`. `LSetProtected` ha visibilità `package` ed estende l'interfaccia `LSet` fornendo la definizione di tutti i metodi usati all'interno di JSetL, ad esempio quelli usati dal risolutore di vincoli.

Le classi concrete sono due: `ConcreteLSet`, che implementa l'interfaccia `LSetProtected` ed è sostanzialmente la classe che fornisce l'implementazione per insiemi JSetL, e `ConcreteMutableLSet`, che implementa `LSetProtected` e `MutableLSet` e può essere considerata come una ulteriore implementazione sia per insiemi JSetL sia per insiemi `java.util.Set`.

La gerarchia proposta, quindi, è la seguente:

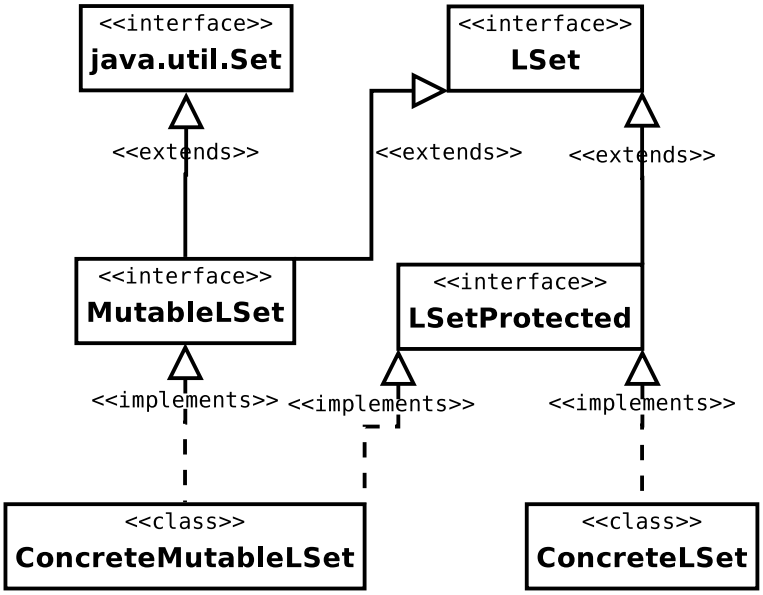


Figura 5.1: Gerarchia di classi per gli "insiemi logici"

## 5.2 Strutture dati

La classe `ConcreteMutableLSet` implementa le interfacce `MutableLSet` e `LSetProtected`. È una classe concreta e quindi può istanziare oggetti (da qui il prefisso `Concrete`).
Un insieme JSetL è considerato un oggetto immutabile, ovvero qualsiasi operazione che ne preveda la modifica in realtà ne restituisce una copia. Un `MutableLSet` invece, analogamente a un insieme Java, può essere modificato (da qui il prefisso `Mutable`).

I metodi ereditati da `java.util.Set`, che modificano l'insieme oggetto di invocazione, sono simulati in [1] da metodi di `LSet` su un oggetto di tipo `ConcreteLSet`. La soluzione è però particolarmente inefficiente; esempi per i metodi di base `add`, `contains` e `remove` sono visti in 6.1.
Per risolvere questo problema la classe `ConcreteMutableLSet` *usa* al suo interno due diversi oggetti per memorizzare gli elementi dell'insieme: `hSet` di tipo `java.util.HashSet` dove, sotto determinate condizioni, i metodi di `java.util.Set` operano direttamente, e `set` di tipo `ConcreteLSet`.
La sostanziale differenza dal lavoro di tesi di Delia Di Giorgio è la presenza di questa doppia rappresentazione. Un oggetto di tipo `ConcreteMutableLSet` ha quindi al suo interno sia un oggetto di tipo `java.util.HashSet` sia uno di tipo `ConcreteLSet` secondo una relazione di tipo has-a.
Per non creare ridondanza e limitare lo spazio occupato, sono previste funzioni di passaggio dalle due rappresentazioni che impostano a `null` l'oggetto non utilizzato. Queste funzioni sono richiamate quando sono invocati alcuni metodi di `java.util.Set`, e la conversione avviene solo se l'insieme oggetto di invocazione è ground. Un obiettivo dell'implementazione è di rendere minimo il numero di passaggi da `hSet` a `set` e viceversa.
Un altro obiettivo dell'implementazione è di cercare di limitare le modifiche al package JSetL. Un principio generale della programmazione ad oggetti infatti afferma che le entità software dovrebbero essere aperte per le estensioni e chiuse alle modifiche (Open-Closed Principle).

In linea con l'implementazione `ConcreteLSet` di `LSet`, sono stati aggiunti alla classe `ConcreteMutableLSet` gli attributi: `name` per il nome dell'insieme, `prefix`, `counter` e `id` per assegnare un nome di default quando non è specificato.

Lo scheletro generale della classe rispecchia lo schema:

```
public class ConcreteMutableLSet implements MutableLSet,
LSetProtected, Serializable, Cloneable {
  private ConcreteLSet set;
  private HashSet hSet;
  private static String prefix = "MutableLSet_";
  private static int counter = 0;
  private int id;
  private String name;
  // COSTRUTTORI
  ...
  // METODI DELL'INTERFACCIA LSetProtected
  ...
  // METODI DELL'INTERFACCIA MutableLSet
  ...
  // METODI DI PASSAGGIO
  ... }
```

## 5.3 Costruttori

Nella classe `ConcreteLSet` sono previsti numerosi costruttori, ad esempio costruttori con parametro array per ogni tipo primitivo.
Per rendere più semplice la comprensione delle differenze tra i diversi modi per creare un insieme da parte dell'utente (gli insiemi prevedono anche i costruttori definiti dalle convenzioni di `java.util.Collection` in 2.1), la

classe `ConcreteMutableLSet` prevede solo una parte di questi costruttori. Tale semplificazione migliora la leggibilità del codice per una successiva manutenzione o riuso.

Nella classe `ConcreteLSet` con il costruttore di default è costruito un `LSet` non inizializzato. In modo analogo il costruttore di default di `Concrete MutableLSet` costruisce un nuovo insieme non inizializzato assegnando a `set` un nuovo insieme non inizializzato `ConcreteLSet` e ponendo `hSet` a `null`. Da notare l'importante differenza con il costruttore di default per una `java.util.Collection` che crea un insieme vuoto.

```
public ConcreteMutableLSet() {
  this.set=new ConcreteLSet();
  this.hSet=null;
  this.setId();
  this.setName(prefix.concat((new Integer(this.id)).
      toString()));
}
```

Di tutti i costruttori in `ConcreteLSet` con parametro un array di elementi di tipo primitivo, ora è previsto solo il costruttore per il tipo `int`.

```
public ConcreteMutableLSet(int[] intarr) {
  if (intarr.length==0) {
    this.set=ConcreteLSet.empty;
    this.hSet=null;
  }
  else {
    this.hSet=new HashSet();
    this.set=null;
    for(int i=0;i<intarr.length;i++)
      this.hSet.add(new Integer(intarr[i]));
    }
```

```
    this.setId();
    this.setName();
}
```

Se l'array specificato è vuoto è assegnata a `set` la costante `ConcreteLSet.empty`.
Si noti la convenzione adottata per la rappresentazione di un insieme vuoto: *un insieme* `ConcreteMutableLSet` *è vuoto se e solo se* il suo attributo `set` *è la costante* `ConcreteLSet.empty`.
Un insieme di elementi interi è ground, quindi gli elementi dell'array specificato sono inseriti iterativamente in `hSet`; `set` è posto a `null`. Si noti che il metodo `add` di `HashSet` non prevede la possibilità di inserire valori di tipo primitivo; i valori sono convertiti dal tipo primitivo con la wrapper class `Integer` (in fase di creazione è specificato nel costruttore di `Integer` il valore da inglobare).

Analogamente a `ConcreteLSet`, anche in `ConcreteMutableLSet`, è previsto il cotruttore con parametri due interi `p` e `q` (di cui è omesso il codice) che costruisce un nuovo insieme i cui elementi sono costituiti da tutti i numeri `int` nell'intervallo chiuso [p,q]. Anche in questo caso se l'intervallo specificato è vuoto, cioè se `p>q`, è creato un insieme vuoto, altrimenti gli interi sono inseriti iterativamente in `hSet`.

Per uniformità con i costruttori degli oggetti `java.util.Collection` è presente anche un costruttore con parametro di tipo `java.util.Collection`.

```
public ConcreteMutableLSet(Collection c)
    throws NotGroundElementException{
  if (c.size()==0) {
    this.set=ConcreteLSet.empty;
    this.hSet=null;
  }
  else {
```

```
    this.set=null;
    this.hSet=new HashSet();
    this.addAll(c);
  }
  this.setId();
  this.setName();
}
```

Anche in questo caso, se la collezione passata è vuota, è assegnata a `set` la costante `ConcreteLSet.empty`; altrimenti è iniziallizzato `hSet` e richiamato il metodo `addAll` che lancia un'eccezione se un elemento della collezione specificata non è ground.

Per ognuno dei precedenti costruttori pubblici ne esiste uno che ha come primo parametro una stringa (di cui è omesso il codice) il quale richiama il relativo costruttore senza stringa e imposta il parametro stringa specificata come nome dell'insieme.

Sono previsti infine due costruttori `protected`, di visibilità interna al package JSetL, uno con parametro `java.util.HashSet`, l'altro con parametro `ConcreteLSet` (di cui è omesso il codice), che creano un nuovo `ConcreteMutableLSet` assegnando rispettivamente a `hSet` oppure a `set` il parametro specificato.

```
private ConcreteMutableLSet(HashSet hs) {
  this.set=null;
  this.hSet=hs;
  this.setId();
  this.setName();
}
```

Si è visto come il costruttore senza parametri crei un insieme non inizializzato. Per costruire un insieme vuoto possiamo sfruttare uno dei costruttori

precedentemente visti passando come parametro un array vuoto, un intervallo vuoto o una collezione vuota. È però previsto un metodo più immediato, analogo a quello previsto per gli `LSet`.
Siccome gli insiemi `ConcreteLSet` sono immutabili, l'insieme vuoto di tipo `ConcreteLSet` è unico e definibile come attributo statico per la classe `ConcreteLSet`; ogni `ConcreteLSet` vuoto ha lo stesso riferimento. Questa soluzione non funziona per gli insiemi `ConcreteMutableLSet` che sono mutabili. È presente un metodo statico per la classe `ConcreteMutableLSet` che ad ogni chiamata restituisce un nuovo insieme vuoto `ConcreteMutableLSet`.

```
public static ConcreteMutableLSet empty() {
  return new ConcreteMutableLSet(ConcreteLSet.empty);
}
```

## 5.4 Metodi di passaggio implementazione insieme

I metodi di passaggio di implementazione insieme sono particolari metodi non visibili all'utente (di visibilità interna alla classe) che convertono la rappresentazione interna da `ConcreteLSet` a `HashSet` e viceversa.
Sono stati costruiti partendo dalle seguenti considerazioni:

- Un `ConcreteMutableLSet` è rappresentabile da `ConcreteLSet`.
- Un `ConcreteMutableLSet` è rappresentato da `HashSet` solo se è ground.

Un `ConcreteMutableLSet` è ground quando, partendo da un insieme ground, sono chiamati soltanto metodi ereditati da `java.util.Set`, oppure metodi ereditati da `LSetProtected` che mantengono l'insieme ground.
Il metodo di passaggio da `ConcreteLSet` a `HashSet` è il seguente (dei restanti per brevità è omesso il listato):

```
private void toHSet throws NotGroundSetException(){
  if(this.hSet==null){ //se l'insieme è già hSet non converte
    if(this.set.isGround()) {
      this.hSet=new HashSet();
```

```
      Vector v=set.toVector();
      Iterator i=v.iterator();
      Object obj;
      while(i.hasNext()) {
        //inserisce gli elementi uno a uno
        obj=i.next();
        //inserisce valore di Lvar
        if(obj instanceof Lvar) hSet.add(rtoHSet(((Lvar)obj)
            .getValue()));
        else if(obj instanceof LSet) hSet
            .add(rtoHSet((LSet)obj));
        else if(obj instanceof Lst) hSet
            .add(rtoHSet((Lst)obj));
        else hSet.add(obj);
      }
    this.set=null;
    this.setId();
  }
  else throw new NotGroundSetException();
  }
}
```

Si noti che se l'insieme oggetto di invocazione ha `hSet` a `null` e non è ground il metodo lancia un'eccezione. Se invece è ground inizializza `hSet` e inserisce iterativamente gli elementi restituiti dal metodo `toVector` di `set` convertiti dalla funzione `rtoHSet()`, e imposta `set` a `null`.

Per convertire ogni elemento dell'insieme oggetto di invocazione a ground, compresi eventuali insiemi annidati, è stato previsto un nuovo metodo, di visibilità interna alla classe:

```
private Object rtoHSet(Object s) { ... }
```

Il metodo esamina ricorsivamente `Lvar`, `LSet` e `Lst` annidati e restituisce un nuovo oggetto dove alle `Lvar` sono sostituiti i valori, e ogni `LSet` è convertito tramite `rtoHSet` a `MutableLSet` rappresentato con `HashSet`.
Il metodo di passaggio da `HashSet` a `ConcreteLSet` è il seguente:

```
private void toSet() { ... }
```

Il metodo, se l'insieme oggetto di invocazione ha `set` a `null`, inizializza `set` con gli elementi dell'array ottenuto da `hSet`, e imposta `hSet` a `null`.

## 5.5 Metodi ereditati da java.util.Set

I metodi ereditati da `java.util.Set` sono parte dei metodi ereditati dall'interfaccia `MutableLSet`.
L'implementazione dei metodi, che possono modificare l'insieme oggetto di invocazione ereditati da `java.util.Set`, ha il seguente comportamento comune: richiama il metodo di passaggio `toHSet` e, se l'insieme oggetto di invocazione è ground, richiama il relativo metodo di `HashSet` sull'oggetto `hSet` invece, se l'insieme oggetto di invocazione non è ground, propaga l'eccezione generata dal metodo di passaggio.
Come esempio è mostrato il listato del metodo `add`:

```
public boolean add(Object o) throws NotGroundSetException,
    NotGroundElementException {
  if(this.hSet==null) this.toHSet();
  if (o instanceof Lvar) { //Lvar non ground
    if (!((Lvar)o).isGround()) throw new
        NotGroundElementException();
    else return this.hSet.add(this.rtoHSet(o));
  }
  else if (o instanceof LSet){
  if (!((LSet)o).isGround()) throw new
        NotGroundElementException();
```

```
    else return this.hSet.add(this.rtoHSet(o));
  }
  else if (o instanceof Lst){
  if (!((Lst)o).isGround()) throw new
        NotGroundElementException();
    else return this.hSet.add(this.rtoHSet(o));
  }
  else return this.hSet.add(o); //oggetto qualsiasi
}
```

Come si vede dal codice è possibile aggiungere oggetti di qualsiasi tipo e in particolare `Lvar`, `LSet` o `Lst`, e se questi sono ground è richiamato il metodo `add` di `hSet` sull'oggetto convertito, altrimenti è lanciata un'eccezione.

## 5.6 Metodi ereditati da LSet

I metodi ereditati da `LSet` sono parte dei metodi ereditati dall'interfaccia `MutableLSet`.
Come nel capitolo precedente possiamo concettualmente dividere i metodi ereditati dall'interfaccia `LSet` in metodi di utilità e metodi di generazione vincoli.
Tutti i metodi di utilità sono stati riscritti cercando di non cambiare la rappresentazione interna dell'insieme se non strettamente necessario; l'esecuzione di un metodo di utilità ereditato da `LSet` non modifica l'insieme oggetto di invocazione, in particolare se un insieme è rappresentato con `HashSet` rimane tale. Come esempio è mostrato il metodo di utilità `toVector`:

```
public Vector toVector() {
  if (this.set==null) {
    Vector v = new Vector(this.hSet);
    return v;
  }
  else return this.set.toVector();
```

```
}
```

Se `set` è a `null`, il metodo crea e restituisce un nuovo `Vector` sfruttando il costruttore di parametro `Collection`, altrimenti richiama il metodo `toVector` su `set`. L'insieme oggetto di invocazione non è modificato, non vi è passaggio di rappresentazione.

Come altro esempio è mostrata la prima parte del metodo `ins`:

```
MutableLSet mlset;

if (hSet==null) { //l'insieme e' ConcreteLSet
  if (this.isEmptySL()) { //l'insieme e' l'insieme vuoto
    if ((o instanceof LSet && !((LSet)o).isGround()) ||
      (o instanceof Lvar && !((Lvar)o).isGround()) ||
      (o instanceof Lst && !((Lst)o).isGround())) {
      Vector v=new Vector();
      v.add(o);
      ConcreteLSet ls=new ConcreteLSet(v,ConcreteMutableLSet
          .empty());
      mlset=new ConcreteMutableLSet(ls);
    }
    else {
      HashSet hs=new HashSet();
      hs.add(rtoHSet(o));
      mlset=new ConcreteMutableLSet(hs);
    }
  }
  else {
  Vector v=new Vector();
  v.add(o);
  ConcreteLSet ls=new ConcreteLSet(v,this);
  mlset=new ConcreteMutableLSet(ls);
```

```
  }
}
```

Se `hSet` è `null` l'insieme oggetto di invocazione è rappresentato con `ConcreteLSet`. Se è l'insieme vuoto e l'oggetto specificato alla `ins` è ground il metodo crea un nuovo insieme rappresentato con `HashSet`; se è l'insieme vuoto e l'oggetto specificato non è ground crea un nuovo insieme rappresentato con `ConcreteLSet`; se non è l'insieme vuoto, crea un nuovo insieme rappresentato con `ConcreteLSet`.
È mostrata la seconda parte del metodo `ins`:

```
else { //l'insieme e' HashSet
  if ((o instanceof LSet && !((LSet)o).isGround()) ||
     (o instanceof Lvar && !((Lvar)o).isGround()) ||
     (o instanceof Lst && !((Lst)o).isGround())) {
      Vector v=new Vector();
      v.add(o);
      v.addAll(this.hSet);
      ConcreteLSet ls=new ConcreteLSet(v,ConcreteMutableLSet.
          empty());
      mlset=new ConcreteMutableLSet(ls);
    }
    else {
      HashSet hs=new HashSet();
      hs.add(rtoHSet(o));
      hs.addAll(this.hSet);
      mlset=new ConcreteMutableLSet(hs);
    }
}
return mlset;
```

Se `set` è `null` l'insieme è rappresentato con `HashSet`. Se l'oggetto specificato alla `ins` è ground il metodo crea un nuovo insieme rappresentato con

`HashSet`; se l'oggetto specificato non è ground crea un nuovo insieme rappresentato con `ConcreteLSet`.
Il metodo `ins` sfrutta diversi metodi e costruttori: il metodo di conversione `rtoHSet` quando aggiunge oggetti ground a `HashSet`, il costruttore per `ConcreteLSet` che specificando un vettore e un insieme `LSetProtected` crea un nuovo insieme `ConcreteLSet` e i costruttori per `ConcreteMutableLSet` di parametro `HashSet` e `ConcreteLSet` per creare il nuovo insieme.

Per quanto riguarda i metodi di generazione vincoli, la loro implementazione è identica a quella degli analoghi metodi in `ConcreteLSet`. Questo è possibile perché `ConcreteMutableLSet` fornisce un'implementazione degli "insiemi logici" e nella risoluzione di vincoli sono utilizzati solamente metodi dichiarati in `LSetProtected`.
Ad esempio il metodo di generazione del vincolo di appartenenza:

```
public Constraint in (LSet set) {
  return new Constraint(this, Enviroment.inCode, set);
}
```

Si noti che la restituzione di un vincolo non modifica l'insieme oggetto di invocazione; al costruttore di `Constraint` sono specificati il codice interno che denota il vincolo di appartenenza e i riferimenti ai due insiemi coinvolti.

## 5.7 Metodi ereditati da LSetProtected

I metodi ereditati dall'interfaccia `LSetProtected` forniscono l'implementazione per i metodi degli insiemi logici di visibilià interna al package.
Analogamente alla riscrittura dei metodi di utilità della classe `LSet`, è minimizzato il numero di passaggi. Come esempio è mostrato il metodo di utilità `sub`:

```
public LSetProtected sub() {
  if (this.set==null) {
```

```
    //se l'insieme è vuoto ritorna null
    if (hSet.size()==0) return null;
    //se l'insieme ha un elemento ritorna l'insieme vuoto
    if (hSet.size()==1) return (LSetProtected)
        ConcreteMutableLSet.empty();
    else return this.subFirst();
  }
  else {
    LSetProtected ls=this.set.sub();
    if(ls==null) return null;
    else return ls;
  }
}
```

Il metodo `sub` è usato nella risoluzione di vincoli nella classe `SolverClass`. Restituisce l'insieme ottenuto dall'estrazione di un elemento dell'insieme, se l'insieme è vuoto restituisce `null`. Si noti come con la chiamata a questo metodo l'insieme oggetto di invocazione non sia modificato.
Un esempio differente è il metodo `setResto`:

```
public void setResto(LSetProtected resto) {
  if (this.set==null) { //l'insieme è hSet
    if (resto.isGround()){ //resto ground
      Vector v=resto.toVector();
      Iterator i=v.iterator();
      while(i.hasNext()){ //inserisce iterativamente elementi
        this.hSet.add(this.rtoHSet(i.next()));
      }
    }
    else {
      this.toSet();
      this.set.setResto(resto);
    }
```

```
  }
  else {
    this.set.setResto(resto);
  }
}
```

Il metodo `setResto` è usato nella risoluzione di vincoli nella classe `Solver Class`. Imposta il campo resto dell'insieme `LSetProtected` oggetto di invocazione con l'insieme `LSetProtected` specificato. Si noti la differenza del campo resto, che può essere anche un `LSetProtected` ground, dalla nozione di "resto" di un insieme logico. Il "resto" di un insieme logico è un insieme non inizializzato se l'insieme non è limitato, altrimenti è l'insieme vuoto. Dall'implementazione del metodo `setResto` si può notare che avviene il passaggio di rappresentazione con il metodo `toSet` se l'insieme oggetto di invocazione ha `set` a `null` e l'insieme specificato non è ground.

# Capitolo 6

# Esempi

Questo capitolo mostra alcuni esempi d'uso della classe `MutableLSet` sia come "insiemi logici" sia come "insiemi standard", confrontando prestazioni e leggibilità del codice nei due casi.
Un primo esempio è dedicato al confronto delle prestazioni delle operazioni di base degli "insiemi standard", `contains`, `remove`, `add`, realizzate tramite simulazione usando un oggetto di tipo `ConcreteLSet` (come erano implementati in [1]) con l'attuale implementazione.
Inoltre dei programmi d'esempio in [5] è riportato `AllPrimes.java` ed è confrontato con un programma che usa gli stessi insiemi `MutableLSet`, ma sfrutta i metodi degli "insiemi standard" Java.
Infine si mostra come per alcuni problemi, con un esempio analogo a 1.4 ma in JSetL, usare gli "insiemi logici" è più intuitivo e rapido, e la soluzione con metodi degli "insiemi standard", seppure sia più efficiente, è meno intuitiva e leggibile.

## 6.1 Contains, remove, add

L'obiettivo dell'esempio è verificare l'efficienza dei metodi di base `contains`, `remove` e `add` per un insieme ground di numero di elementi crescente.
Per effettuare il test sono stati creati un insieme `MutableLSet ms` e un insieme `LSet LS0`. Un insieme `LSet` non fornisce questi metodi; i metodi sono

simulati.
Di seguito è riportato il listato della simulazione del metodo `contains`, con `LS0` oggetto di tipo `LSet` e `X` oggetto di tipo `Lvar`:

```
boolean contains=false;
try{
  solver.add(X.in(LS0));
  solver.solve();
  contains=true;
}
catch(Failure F){}
```

Di seguito è riportato il listato della simulazione del metodo `remove`, con `LS0` oggetto di tipo `LSet` e `X` oggetto di tipo `Lvar`:

```
boolean remove=false;
LSet LS1 = new ConcreteLSet();
try{
  solver.add(LS0.less(X,LS1));
  solver.solve();
  remove=true;
  LS0=LS1;
}
catch(Failure F){}
```

Di seguito è riportato il listato della simulazione del metodo `add`, con `LS0` oggetto di tipo `LSet` e `X` oggetto di tipo `Lvar`:

```
boolean add=false;
int size=LS0.size();
LS0=LS0.ins(X);
if (size!=LS0.size()) add=true;
```

Di seguito è riportato il listato dei tre metodi sull'oggetto `ms`, con `X` oggetto di tipo `Lvar`:

```
ms.contans(X);
ms.remove(X);
ms.add(X);
```

Come visto dall'implementazione di `MutableLSet` nella sezione 5.5 e dalle caratteristiche di `HashSet` nella sezione 2.5, i metodi `contains`, `add` e `remove` hanno costo medio costante. La simulazione del metodo `add` avviene con il metodo di utilità `ins`.
Di seguito è riportato il codice del metodo `ins` per la classe `ConcreteLSet`:

```
public LSet ins(Object o) {
  if(this.equ == null) {
    Vector v = new Vector();
    v.add(o);
    LSet s = new ConcreteLSet(v,this);
    return s;
  }
  else return this.equ.ins(o);
}
```

Il numero di elementi dell'insieme oggetto di invocazione non influisce sull'operazione di inserimento; di fatto è costruito un vettore di un elemento l'oggetto specificato, e creato un nuovo insieme `ConcreteLSet` che ha per campo lista il riferimento al vettore e per campo resto il riferimento all'insieme oggetto di invocazione.
Per effettuare il test sono assegnati ai due insiemi gli elementi di un intervallo `[l,u]`, con `l=0` e `u` che assume valori interi nell'intervallo [100,2000] con passo 100. Per la misurazione del tempo in millisecondi è stata utilizzata la classe `java.util.GregorianCalendar`. Per ogni intervallo sono stati assegnati a una `Lvar X` dieci valori casuali, nell'intervallo [l,u] usando la classe `java.util.Random` e misurato il tempo medio necessario per ogni operazione in ogni intervallo.
Di seguito sono riportati i risultati ottenuti. I risultati mostrano come la

risoluzione di vincoli sia particolarmente inefficiente. Sono stati omessi i tempi del metodo `add` e della sua simulazione perché inferiori a un millisecondo.

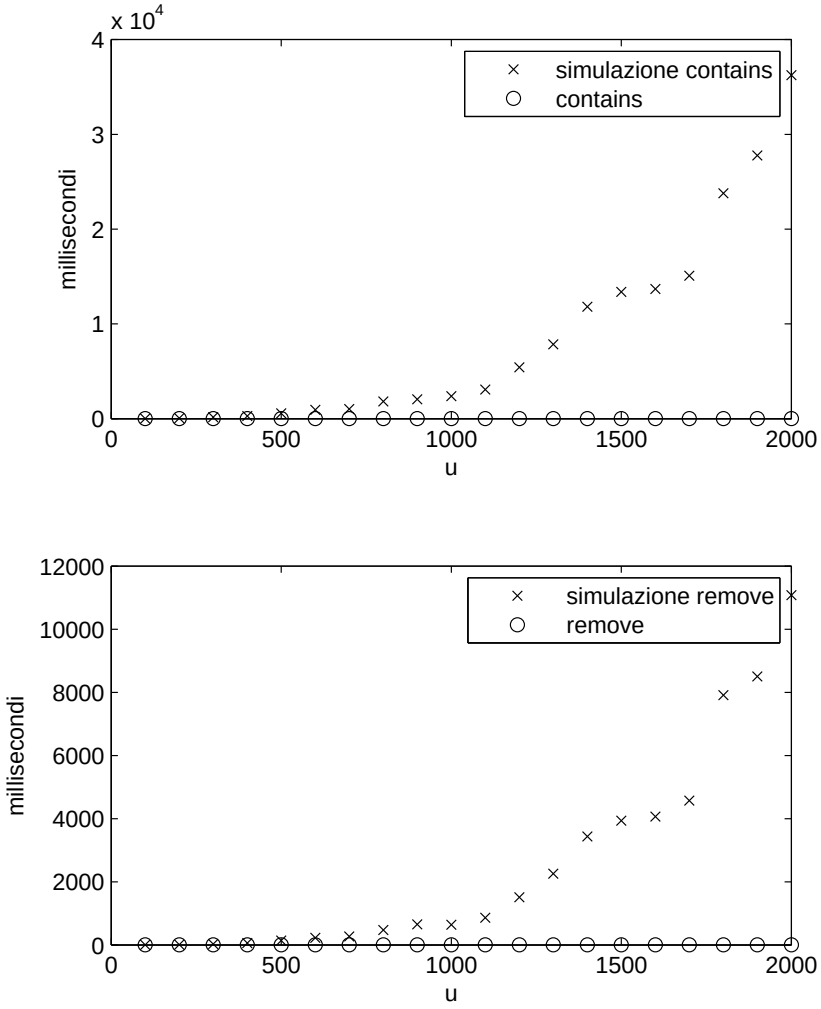


Figura 6.1: Confronto tra metodi nuovi e loro simulazione con LSet

## 6.2 AllPrimes

L'obiettivo del programma preso tra gli esempi JSetL è generare l'insieme di tutti i numeri primi minori o uguali a un naturale arbitrario.
L'esempio è stato modificato perchè risolva il problema anche in modo imperativo trattando l'insieme da trovare come `Set` di Java, confronti gli insiemi trovati e riporti il tempo impiegato in millisecondi.
Di seguito è riportato il listato della funzione che risolve il problema in modo dichiarativo sfruttando i metodi JSetL, dove `Solver` è il risolutore di vincoli.

```
public static MutableLSet primes(int N){
  MutableLSet ret=ConcreteMutableLSet.empty();
  if(N<2) return ret;
  Lvar L = new Lvar();
  Solver.add(L.in(new ConcreteMutableLSet(2,N)));
  try{
    is_prime(L);
    ret=Solver.mutableSetof(L);
  }
  catch(Failure F){}
  return ret;
}
public static void is_prime(Lvar L) throws Failure {
  Lvar X = new Lvar();
  Lvar M = new Lvar();
  MutableLSet s=new ConcreteMutableLSet();
  Solver.add(M.eq(L.sub(1)));
  Solver.add(s.interv(2,M));
  Solver.forall(X,s,(L.mod(X)).neq(0));
  Solver.solve();
  return;
}
```

Il metodo `primes` restituisce l'insieme `MutableLSet` dei numeri primi minori o uguali all'intero passato `N`. Se `N` è minore di 2, `primes` restituisce l'insieme vuoto. Altrimenti è dichiarata una variabile logica `L` non inizializzata. Quando l'oggetto `Solver` invoca il metodo `add` con parametro `L.in(new ConcreteMutableLSet(2,N))` è aggiunto al Constraint Store il vincolo di appartenenza di `L` all'insieme che ha come elementi gli interi dell'intervallo di estremo inferiore 2 e estremo superiore il valore di `N`. Il vincolo è soddisfatto se `L` appartiene all'insieme; con `L` non inizializzata quando l'espressione è valutata equivale ad assegnare in modo non deterministico un elemento dell'insieme a `L`. Nel metodo `primes` infine è richiamato il metodo `is_prime` su `L`. Il blocco `try` ha lo scopo di catturare `Failure` solo per esigenza di compilazione.

Il metodo `is_prime` inserisce nel Constraint Store i vincoli perchè `L` sia un numero primo.

Sono dichiarate le variabili logiche non inizializzate `X` e `M`, e l'insieme logico non inizializzato `s`. L'oggetto `Solver` inserisce con metodo `add` il vincolo `M.eq(L.sub(1))` nell'attuale Constraint Store. Tale vincolo impone che `M` sia uguale al valore di `L-1`. Di nuovo l'oggetto `Solver` invoca il metodo `add`, con parametro `s.interv(2,M)`. Tale vincolo è soddisfatto se l'insieme `s` ha valori dell'intervallo di interi di estremo inferiore 2 e estremo superiore il valore di `M`. L'invocazione del metodo `forall` aggiunge al Constraint Store il vincolo `(L.mod(X)).neq(0)`, per ogni `X` appartenente all'insieme `s`. Tale vincolo è soddisfatto se per ogni elemento `X` di `s` il resto della divisione del valore di `L` con l'elemento non è zero.

Il metodo `mutableSetOf` invocato sull'oggetto `solver` restituisce l'insieme delle soluzioni. Quando è invocato, il risolutore di vincoli controlla se i vincoli nel Constraint Store sono soddisfatti; se lo sono l'invocazione del metodo crea un elemento nell'insieme `MutableLSet` delle soluzioni. Altrimenti se un vincolo non è soddisfatto si ha il backtracking e il calcolo torna indietro al punto di scelta più vicino. In questo caso si torna a quello creato dal vincolo `L.in(new ConcreteMutableLSet(2,N))`. La sua esecuzione lega in modo

non deterministico `L` con ogni elemento dell'insieme. Se non esistessero soluzioni `primes` terminerebbe sollevando un'eccezione `Failure`.

Per risolvere il problema il modo dichiarativo risulta particolarmente inefficiente; sia `N` l'intero passato, `L` è legata ai valori da 2 a `N`, `N`-1 valori, e per ognuno è creato un insieme di `L`-1 elementi, oltre alle operazioni di sottrazione per determinare l'estremo superiore dell'insieme creato e `L`-1 divisioni.

Di seguito è riportato il listato della stessa funzione che risolve il problema in modo imperativo utilizzando sempre un `MutableLSet` ma operando su di esso tramite metodi ereditati dagli insiemi Java.

```
public static MutableLSet primes(int N) {
  MutableLSet ml=ConcreteMutableLSet.empty();
  if(N<2) return ml;
  for(int i=2;i<=N;i++){
    Iterator j=ml.iterator();
    boolean c=false;
    while(j.hasNext()&&!c){
      if (i%((Integer)j.next()).intValue()==0) c=true;
    }
    if (!c) ml.add(i);
    }
  return ml;
}
```

La funzione crea un nuovo insieme MutableLSet `ml` vuoto. Se l'intero passato `N` è minore di 2 restituisce tale insieme. Se `N` è maggiore o uguale a due, il ciclo `for` itera `i` da 2 fino a `N` compreso e verifica se il valore corrente `i` è un numero primo; sfrutta il metodo `iterator` sugli elementi di `ml` che è l'insieme dei numeri primi minori di `i`, e se il resto della divisione con `i` per ogni elemento dell'insieme `ml` è diverso da zero, aggiunge con metodo `add` il valore `i` a `ml`.

Per risolvere il problema la strategia proposta è più efficiente della precedente con risolutore di vincoli, pur essendo di complessità in tempo polinomiale; il ciclo esterno itera `N`-1 volte, è creato un oggetto di tipo `Iterator` e il ciclo interno è eseguito, nel caso pessimo cioè l'elemento corrente `i` è un numero primo, tante volte quanto il numero dei numeri primi minori di `i`. Come visto dall'implementazione di `MutableLSet` nella sezione 5.5 e dalle caratteristiche di `HashSet` nella sezione 2.5 il metodo `add` ha costo medio costante, e l'iterazione sugli elementi ha costo lineare.

La differenza di prestazioni è stata verificata con numero `N` equispaziato nell'intervallo [50,500] con passo 50.

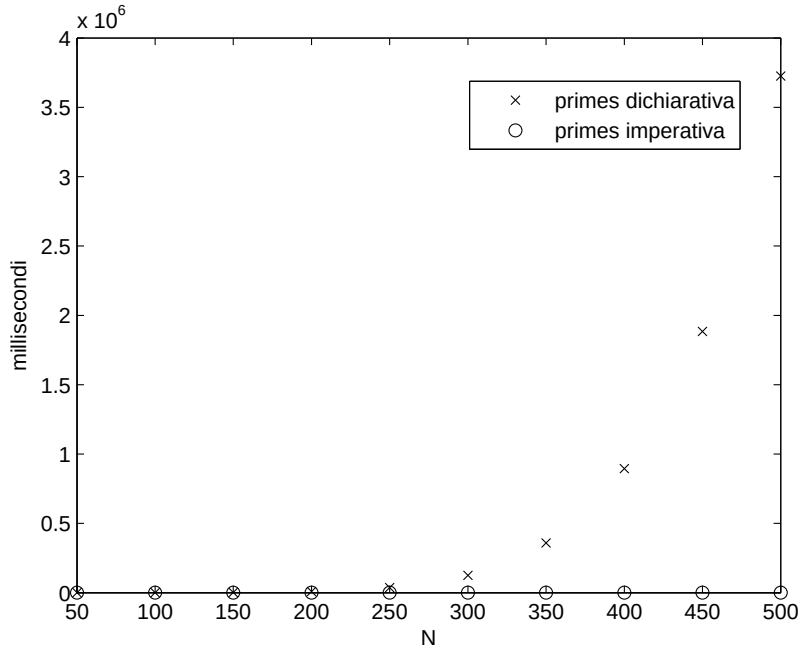


Figura 6.2: Rappresentazione del confronto dei due metodi

Dai risultati ottenuti si noti come per il metodo `primes` dichiarativo le pre-

stazioni decadano velocemente al crescere di `N`, mentre con il metodo `primes` iterativo il tempo di esecuzione per questi valori di `N` rimane nell'ordine delle unità di millisecondi. Con valori superiori di `N` nell'ordine delle migliaia, l'esecuzione di `primes` dichiarativo si arresta per termine dello spazio heap di Java, mentre la versione iterativa può ancora essere utilizzata.

## 6.3 Chiusura transitiva

Negli esempi precedenti si è notata la scarsa efficienza delle soluzioni dichiarative, ma non sempre l'efficienza è ciò che interessa.
Si può risolvere un problema in modo dichiarativo, rapidamente e in modo intuitivo, e cercare un algoritmo più efficiente in un secondo momento.

Lo scopo di questo esempio è, data una rete dove i collegamenti tra nodi hanno una direzione, determinare i nodi raggiungibili da ogni nodo. Il problema è analizzato in 1.4 e ne è data una soluzione in linguaggio SETL in cui il problema è risolto cercando la chiusura transitiva della relazione binaria associata alla rete.
Una relazione binaria in JSetL può essere rappresentata da un insieme `MutableLSet` i cui elementi sono `Lst` di due elementi. Ad esempio la relazione `E={(a,b),(b,c),(c,d)}` sull'insieme `V={a, b, c, d}` è rappresentabile con un insieme costrtuito con una sequenza di comandi del tipo:

```
Lst C=Lst.empty.insn('a').insn('b');
Lst D=Lst.empty.insn('b').insn('c');
Lst E=Lst.empty.insn('c').insn('d');
MutableLSet E=ConcreteMutableLSet.empty().ins(C).ins(D).ins(E);
```

Di seguito è riportato il listato della funzione che restituisce la chiusura transitiva della relazione rappresentata da un insieme `MutableLSet`, dove `Solver` è il risolutore di vincoli:

```
public static MutableLSet transitiveClosure(MutableLSet T) {
  Lvar A = new Lvar();
  Lvar B = new Lvar();
  Lvar C = new Lvar();
  Lst arco1 = Lst.empty.insn(A).insn(B);
  Lst arco2 = Lst.empty.insn(B).insn(C);
  Lst arco3 = Lst.empty.insn(A).insn(C);
  Solver.add(arco1.in(T));
  Solver.add(arco2.in(T));
  Solver.add(arco3.nin(T));
  try {
    Solver.solve();
    T=T.ins(arco3);
    return transitiveClosure(T);
  }
  catch (Failure X){
    return T;
  }
}
```

Sono dichiarate tre variabili logiche non inizializzate, `A`, `B` e `C`.
Il problema è ricondotto alla ricerca dell'esistenza di tre liste di due elementi, `arco1` e `arco2` che appartengano a `T` e `arco3` che non appartenga a `T`. `arco1` e `arco2` devono avere rispettivamente la seconda e la prima componente uguali cioè `B`; `arco3` deve avere la prima componente di `arco1` cioè `A` e la seconda componente di `arco2` cioè `C`. Sono aggiunti al Constraint Store i vincoli necessari.
Se non esiste `arco3` il metodo `solve` fallisce e ed è restituito l'insieme corrente `T` che l'insieme che rappresenta la chiusura transitiva, altrimenti è inserito, con metodo `ins` che crea un nuovo insieme, `arco3` a `T` ed è restituito il metodo richiamato ricorsivamente sul nuovo insieme `T`.

Di seguito è riportato la rappresentazione dello stesso grafo dell'esempio in 1.4 con la soluzione in JSetL:

```
...
Lst A=Lst.empty.insn('a').insn('e');
Lst B=Lst.empty.insn('e').insn('f');
Lst F=Lst.empty.insn('f').insn('b');
Lst C=Lst.empty.insn('c').insn('g');
Lst D=Lst.empty.insn('g').insn('d');
Lst E=Lst.empty.insn('d').insn('g');

MutableLSet R = ConcreteMutableLSet.empty().ins(A).ins(B).
    ins(C).ins(D).ins(E).ins(F);

transitiveClosure(R).output();
...
```

Il frammento di programma produce il seguente output:

```
MutableLSet_9713 = {[a, b],[c, d],[d, d],[d, g],[c, g],[a, e],
    [e, f],[g, d],[f, b],[g, g],[e, b],[a, f]}
```

Questa soluzione ha diverse analogie con la soluzione in 1.4. Analogie che riguardano le strutture dati usate per la rappresentazione della rete (insiemi e liste), l'approccio dichiarativo (sono specificate le proprietà degli archi, non come ottenerli), e le operazioni sulle strutture dati coinvolte (il comando `WITH` che restituisce un nuovo insieme SETL è analogo al metodo `ins` che restituisce un nuovo insieme JSetL).

Di seguito è riportato il listato della stessa funzione che risolve il problema in modo imperativo:

```
public static MutableLSet transitiveClosure(MutableLSet R) {
  MutableLSet T=ConcreteMutableLSet.empty();
```

```
  MutableLSet Rk;
  MutableLSet Rk1=(MutableLSet)R.clone();
  while (Rk1.size()>0){
    T.addAll(Rk1);
    Rk=(MutableLSet)Rk1.clone();
    Rk1.clear();
    Iterator i=R.iterator();
    while (i.hasNext()){
      Lst temp=(Lst)i.next();
      Object A=temp.get(0);
      Object B=temp.get(1);
      Iterator p=Rk.iterator();
      while (p.hasNext()){
        Lst temp2=(Lst)p.next();
          if (!temp.equals(temp2)){
          Object C=temp2.get(0);
          Object D=temp2.get(1);
          if (B.equals(C)){
            Lst arco3=Lst.empty.insn(A).insn(D);
            if(!T.contains(arco3)) Rk1.add(arco3);
          }
        }
      }
    }
  }
  return T;
}
```

Nella prima linea è dichiarato `T` come nuovo `MutableLSet` vuoto, l'insieme restituito successivamente come chiusura transitiva.
Nella seconda linea è dichiarato `Rk` oggetto di tipo `MutableLSet`, l'insieme degli archi inseriti in `T` determinati al passo precedente dall'algoritmo.

Nella terza è assegnato a `Rk1` oggetto di tipo `MutableLSet` una copia dell'insieme passato come argomento `R`, l'insieme dei nuovi archi da inserire in `T`.
Il ciclo esterno `while` itera fino a che ci sono nuovi archi da inserire. Sono inseriti gli elementi di `Rk1` in `T`, è assegnato a `Rk` un clone di `Rk1` e a `Rk1` sono rimossi tutti gli elementi.
Il secondo ciclo `while` itera per gli elementi della relazione iniziale `R`: è estratta la lista elemento corrente e le sue componenti `A` e `B`.
Il ciclo più interno itera per gli elementi inseriti al passo precedente, cioè per gli elementi dell'insieme `Rk`: è estratta la lista corrente e le sue componenti `C` e `D`.
Se `B` è equivalente a `C` e la lista che ha come prima componente `A` e come seconda componente `D` non è già presente in T la lista è aggiunta all'insieme `Rk1`. Gli elementi dell'insieme `Rk1` sono elementi da aggiungere a `T` e sono aggiunti al passo successivo.

Si noti come la soluzione dichiarativa è più leggibile e chiara. Trovare un algoritmo che passo-passo risolva il problema, anche se in modo più efficiente, non è così semplice e la soluzione trovata non è di facile lettura.

# Capitolo 7

# Conclusioni e lavori futuri

In questo lavoro *abbiamo realizzato un'unica interfaccia che racchiude i metodi degli* "insiemi standard" `java.util.Set` *e i metodi degli* "insiemi logici" `LSet` senza modificare il package JSetL.

Abbiamo creato l'interfaccia `MutableLSet` e una sua implementazione, la classe `ConcreteMutableLSet`, che sfrutta al suo interno un insieme `HashSet` di Java. Abbiamo quindi rivisto l'implementazione di tutti i metodi di utilità perché operino con questa nuova rappresentazione interna.
Il risultato ottenuto con questa nuova implementazione è un miglioramento dell'efficienza dei metodi di utilità per insiemi ground; miglioramento che abbiamo verificato con i diversi esempi riportati.
Infine abbiamo verificato il funzionamento di tutti i metodi, di utilità e di generazione di vincoli, applicati ai nuovi insiemi `MutableLSet`, e ne abbiamo scritto la documentazione utente in stile Javadoc.

Un lavoro futuro analogo può essere iniziato per le "liste standard" Java e le "liste logiche" JSetL, creando un'astrazione unica `MutableLst` che permetta di operarare con metodi delle liste `java.util.List` e delle liste `Lst`. La gerarchia di classi proposta è analoga a quella vista nel lavoro di tesi.

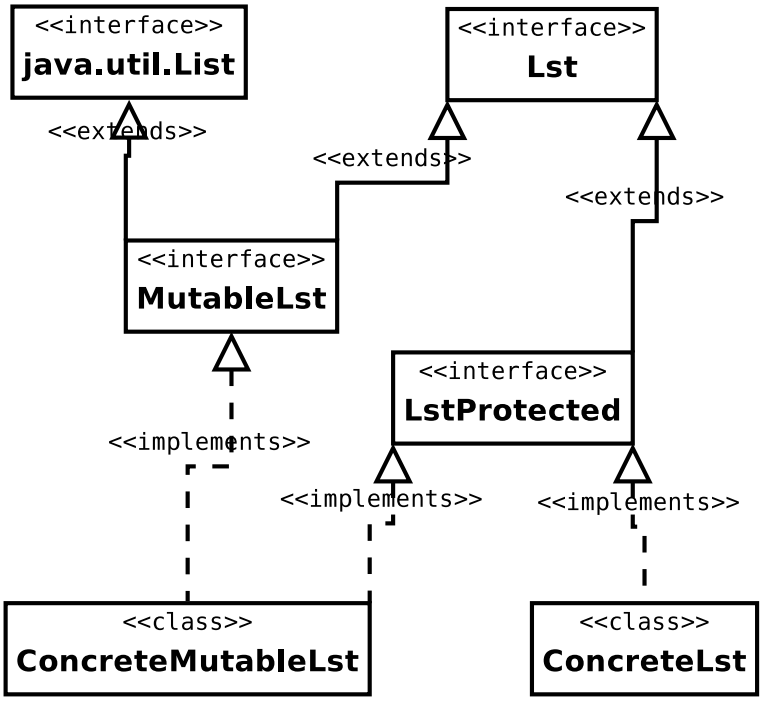


Figura 7.1: Una possibile gerarchia di classi per le liste logiche

Un ulteriore lavoro futuro è suggerito dall'introduzione dalla versione 1.5 di JDK delle classi parametriche. Ad esempio le classi parametriche permettono di specificare il tipo degli elementi per i contenitori della classe `Collection` ampiamente usati nel package JSetL.
Un tipico esempio di utilizzo della classe `Set` senza ricorrere alle classi parametriche potrebbe essere:

```
Set intSet=new HashSet();
intSet.add(new Integer(0));
Integer x=(Integer)intSet.iterator().next();
```

Si ponga l'attenzione sul cast nella terza linea dell'esempio. Tipicamente un programmatore conosce il tipo di dati che sarà inserito in un contenitore, ma il cast è necessario. Il compilatore infatti garantisce che dal metodo `next` dell'iteratore sull'insieme sia restituito un `Object`; per assegnare il valore a una variabile di tipo `Integer` è richiesto il cast, verificato a tempo di esecuzione del programma. L'idea che sta alla base dell'uso delle classi parametriche è imporre che il contenitore contenga un particolare tipo di dati. L'esempio precedente riscritto con l'uso delle classi parametriche diventa:

```
Set<Integer> intSet=new HashSet<Integer>();
intSet.add(new Integer(0));
Integer x=intSet.iterator().next();
```

Si noti dalla dichiarazione dell'insieme che non si tratta di un `Set` arbitrario, ma di un `Set` di `Integer`, scritto `Set<Integer>`. `Set` è infatti una generica interfaccia che prende come parametro il tipo di dati, in questo caso `Integer`; è specificato il tipo anche nella creazione dell'oggetto `new HashSet<Integer>()`. Una importante differenza tra le due versioni è che non è più necessario il cast nella terza linea, il compilatore ora garantisce la correttezza del tipo di dati contenuti in `intSet` per ogni suo uso.
Le classi parametriche, soprattutto per grandi programmi, garantiscono una

migliore leggibilità e robustezza del codice, quindi un lavoro futuro può essere un aggiornamento del package JSetL in questo senso. Impostare a priori il tipo introduce una perdita di generalità; in JSetL è possibile costruire insiemi eterogenei. Occorre verificare se questa perdita di generalità sia evitabile o poco influente. Una guida all'uso e alla creazione di classi parametriche è rimandata alla bibliografia [11].

# Appendice A

# MutableLSet

Di seguito sono elencati i costruttori e i metodi visibili all'utente dell'interfaccia MutableLSet.
Più precisamente si ha l'elenco dei costruttori in appendice A.1, i metodi di utilità in appendice A.2 e i metodi di generazione di vincoli in appendice A.3.

## A.1 Costruttori

Si considera `c` un oggetto di tipo `java.util.Collection`, `intarr` un array di tipo `int`, `p` e `q` di tipo `int`, `name` di tipo `java.lang.String`.

| utilizzo | descrizione |
|---|---|
| `ConcreteMutableLSet()` | `Costruisce un nuovo insieme non inizializzato.` |
| `ConcreteMutableLSet(c)` | `Costruisce un nuovo insieme contenente gli elementi nella collezione specificata (eccezione se un elemento della collezione non è ground, costruisce un insieme vuoto se la collezione è vuota).` |

| utilizzo | descrizione |
|---|---|
| `ConcreteMutableLSet(intarr)` | `Costruisce un nuovo insieme contenente gli elementi nell'array di interi specificato.` |
| `ConcreteMutableLSet(p, q)` | `Costruisce un nuovo insieme i cui elementi sono costituiti da tutti i numeri interi nell'intervallo chiuso [p,q].` |
| `ConcreteMutableLSet(name)` | `Costruisce un nuovo insieme non inizializzato, di nome la stringa specificata.` |
| `ConcreteMutableLSet(name, c)` | `Costruisce un nuovo insieme contenente gli elementi nella collezione specificata, di nome la stringa specificata (eccezione se un elemento della collezione non è ground, costruisce un insieme vuoto se la collezione è vuota).` |
| `ConcreteMutableLSet(name, intarr)` | `Costruisce un nuovo insieme contenente gli elementi nell'array di interi specificato, di nome la stringa specificata.` |

| utilizzo | descrizione |
|---|---|
| `ConcreteMutableLSet(name, p, q)` | `Costruisce un nuovo insieme i cui elementi sono costituiti da tutti i numeri interi nell'intervallo chiuso [p,q], di nome la stringa specificata.` |

## A.2 Metodi di utilità

Si considera `s` oggetto d'invocazione di tipo `MutableLSet`, `o` un oggetto di tipo `java.Object`, `c` oggetto di tipo `java.util.Collection`, `arr` un array di `java.Object`, `name` di tipo `java.lang.String`

| utilizzo | descrizione |
|---|---|
| `s.add(o)` | `Aggiunge l'oggetto specificato a questo insieme; restituisce vero se l'oggetto non è già presente (eccezione se questo insieme non è ground, eccezione se l'oggetto non è ground).` |
| `s.addAll(c)` | `Aggiunge tutti gli elementi della collezione specificata a questo insieme se non sono già presenti (eccezione se questo insieme non è ground, eccezione se un elemento della collezione non è ground).` |
| `s.addAll(intarr)` | `Aggiunge tutti gli elementi dell'array specificato a questo insieme se non sono già presenti (eccezione se questo insieme non è ground).` |

| utilizzo | descrizione |
|---|---|
| `s.clear()` | Rimuove tutti gli elementi da questo insieme; l'insieme diventa l'insieme vuoto (eccezione se questo insieme non è bound). |
| `s.clone()` | Crea e ritorna una copia superficiale di questo insieme. |
| `s.contains(o)` | Ritorna vero se l'oggetto specificato è presente in questo insieme (eccezione se questo insieme non è ground). |
| `s.containsAll(c)` | Ritorna vero se tutti gli elementi della collezione specificata sono presenti in questo insieme (eccezione se questo insieme non è ground, eccezione se un elemento della collezione non è ground). |
| `s.equals(o)` | Confronta l'oggetto specificato con questo insieme e ritorna vero se sono equivalenti. |
| `s.get(int i)` | Ritorna l'i-esimo elemento del Vector costruito a partire dagli elementi di questo insieme. |
| `s.getClass()` | Ritorna il tipo reale di questo insieme. |
| `s.getFirst()` | Ritorna il primo elemento (secondo sintassi) di questo insieme, se l'insieme e‘ vuoto ritorna null. |
| `s.getName()` | Ritorna il nome dell'insieme. |
| `s.getRest()` | Ritorna il ‘‘resto’’ di questo insieme (ovvero un riferimento a un insieme non inizializzato), oppure l'insieme vuoto se questo insieme non ha ‘‘resto’’. |
| `s.hashCode()` | Ritorna il codice hash di questo insieme. |
| `s.ins(o)` | Restituisce l'insieme ottenuto aggiungendo l'oggetto specificato a questo insieme; lascia invariato questo insieme. |

| utilizzo | descrizione |
|---|---|
| `s.insAll(c)` | Restituisce l'insieme ottenuto aggiungendo tutti gli elementi della collezione specificata a questo insieme; lascia invariato questo insieme. |
| `s.insAll(intarr)` | Restituisce l'insieme ottenuto aggiungendo tutti gli elementi dell'array specificato a questo insieme; lascia invariato questo insieme. |
| `s.isBound()` | Ritorna vero se questo insieme è limitato (ovvero questo insieme non ha ‘‘resto’’). |
| `s.isEmpty()` | Ritorna vero se questo insieme non ha elementi. |
| `s.isGround()` | Ritorna vero se questo insieme è ground (ovvero non contiene variabili non inizializzate ed è limitato). |
| `s.isKnown()` | Ritorna vero se questo insieme è inizializzato, falso altrimenti. |
| `s.iterator()` | Ritorna un iteratore sugli elementi di questo insieme (eccezione se questo insieme non è ground). |
| `s.label()` | Applica la procedura di labeling singolarmente ad ognuna delle variabili contenute in questo insieme. |
| `s.normalizeSet()` | Ritorna un insieme ottenuto da questo insieme eliminando tutti gli eventuali elementi con lo stesso valore. |
| `s.occurs(o)` | Ritorna vero se l'oggetto specificato è presente in questo insieme oppure in un insieme annidato in ogni livello di questo insieme (ignora un eventuale ‘‘resto’’, ritorna falso se questo insieme è non inizializzato). |

| utilizzo | descrizione |
|---|---|
| `s.output()` | Stampa il nome ed il valore di questo insieme. |
| `s.print()` | Stampa il valore di questo insieme se l'insieme è inizializzato, altrimenti stampa il nome di questo insieme preceduto da ``_''. |
| `s.read()` | Ritorna l'insieme costituito dall'insieme di elementi letti dallo stream di input; l'insieme fornito in input deve essere delimitato da graffe e gli elementi separati da virgole; gli elementi ammessi sono numeri, caratteri delimitati da apici o stringhe delimitate da apici doppi. |
| `s.remove(o)` | Rimuove l'oggetto specificato da questo insieme se presente (eccezione se questo insieme non è ground, eccezione se l'oggetto non è ground). |
| `s.removeAll(c)` | Rimuove da questo insieme tutti gli elementi contenuti nella collezione specificata (eccezione se questo insieme non è ground, eccezione se un elemento della collezione non è ground). |
| `s.retainAll(c)` | Rimuove tutti gli elementi di questo insieme che non fanno parte della collezione specificata (eccezione se questo insieme non è ground, eccezione se un elemento della collezione non è ground). |
| `s.setName(name)` | Imposta il nome di questo insieme con la stringa specificata. |

| utilizzo | descrizione |
|---|---|
| `s.size()` | Ritorna il numero di oggetti di questo insieme (ignora un eventuale ‘‘resto’’, ritorna il numero di oggetti di questo insieme se questo insieme non è ground). |
| `s.toArray()` | Ritorna un array contenente tutti gli elementi di questo insieme (ignora un eventuale ‘‘resto’’, ritorna un array vuoto se questo insieme è non inizializzato). |
| `s.toArray(arr)` | Copia nell’array specificato tutti gli elementi di questo insieme (ignora un eventuale ‘‘resto’’, lascia invariato l’array specificato se questo insieme è non inizializzato). |
| `s.toString()` | Ritorna una stringa di sintassi corrispondente al valore di questo insieme. |
| `s.toVector()` | Ritorna un Vector contenente tutti gli elementi di questo insieme (ignora un eventuale ‘‘resto’’, ritorna un Vector vuoto se questo insieme è non inizializzato). |

## A.3 Metodi di generazione di vincoli

Si considera `s` oggetto di invovazione di tipo `MutableLSet`, `o` un oggetto di tipo `java.Object`, `ls`, `ls1`, `ls2` oggetti di tipo `LSet`, `lv`, `lv1`, `lv2` oggetti di tipo `Lvar`. Tutti i metodi di generazione di vincoli di parametri `ls`, `ls1`, `ls2` prevedono che questi possano essere `Lvar` con all’interno `LSet`.

| utilizzo | descrizione |
|---|---|
| `s.allDifferent()` | Costruisce il vincolo che impone che tutti gli elementi di s siano diversi fra loro, implementato con decomposizione binaria. |
| `s.allDistinct()` | Costruisce il vincolo che impone che tutti gli elementi di s siano diversi fra loro, implementato come vincolo globale. |
| `s.differ(ls1,ls2)` | Costruisce la congiunzione di vincoli differenza, il vincolo è vero se questo insieme è uguale alla differenza degli insiemi specificati. |
| `s.disj(ls)` | Costruisce il vincolo di disgiunzione, il vincolo è vero se questo insieme e l'insieme specificato sono disgiunti. |
| `s.eq(o)` | Costruisce il vincolo di uguaglianza, il vincolo è vero se questo insieme è uguale all'oggetto specificato. |
| `s.in(ls)` | Costruisce il vincolo di appartenenza, il vincolo è vero se questo insieme appartiene all'insieme specificato. |
| `s.inters(ls1,ls2)` | Costruisce il vincolo intersezione, il vincolo è vero se questo insieme è uguale all'intersezione degli insiemi specificati. |
| `s.interv(lv1,lv2)` | Costruisce il vincolo che impone che questo insieme abbia come elementi gli elementi dell'intervallo limitato inferiormente dal valore intero di lv1 e superiormente dal valore intero di lv2. |

| utilizzo | descrizione |
|---|---|
| `s.less(lv,ls)` | Costruisce la congiunzione di vincoli less, il vincolo è vero se l'insieme specificato è questo insieme privato dell'elemento della variabile logica specificata e l'insieme specificato non contiene la variabile logica specificata. |
| `s.ndiffer(ls1,ls2)` | Costruisce la congiunzione di vincoli differenza, il vincolo è vero se questo insieme è diverso dalla differenza degli insiemi specificati. |
| `s.ndisj(ls)` | Costruisce il vincolo di non disgiunzione, il vincolo è vero se questo insieme e l'insieme specificato non sono disgiunti. |
| `s.neq(o)` | Costruisce il vincolo di disuguaglianza, il vincolo è vero se questo insieme è diverso dall'oggetto specificato. |
| `s.nin(ls)` | Costruisce il vincolo di non appartenenza, il vincolo è vero se questo insieme non appartiene all'insieme specificato. |
| `s.ninters(ls1,ls2)` | Costruisce il vincolo non intersezione, il vincolo e‘ vero se questo insieme e‘ diverso dall'intersezione degli insiemi specificati. |
| `s.nsubset(ls)` | Costruisce il vincolo di non inclusione, il vincolo è vero se questo insieme non è sottoinsieme dell'insieme specificato. |
| `s.nunion(ls1,ls2)` | Costruisce il vincolo di non unione, il vincolo è vero se questo insieme non è unione degli insiemi specificati. |

| utilizzo | descrizione |
|---|---|
| `s.size(lv)` | `Costruisce il vincolo cardinalità, se lv ha valore n intero non negativo` \|s\|=n `o se la variabile lv è unknown, eccezione se lv ha valore intero negativo o valore non intero.` |
| `s.size(n)` | `Costruisce il vincolo cardinalità` \|s\|=n, `eccezione se n è intero negativo.` |
| `s.subset(ls)` | `Costruisce il vincolo di inclusione, il vincolo è vero se questo insieme è sottoinsieme dell'insieme specificato.` |
| `s.union(ls1,ls2)` | `Costruisce il vincolo di unione, il vincolo è vero se questo insieme è l'unione degli insiemi specificati.` |

# Bibliografia

[1] Delia Di Giorgio
*Gestione di insiemi e operazioni insiemistiche in Java tramite l'integrazione tra la libreria JSetL e l'interfaccia Set di Java*
Tesi di laurea triennale, Università di Parma, 2006

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

[3] Robert B.K. Dewar
*The SetL Programming Language*
New York University Computer Science department, 1979

[4] Agostino Dovier, Roberto Giacobazzi
*Fondamenti dell'Informatica: Linguaggi Formali, Calcolabilità e Complessità*
Università degli Studi di Udine, 2007

[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/*

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
*Introduction to Algorithms*
The Massachussetts Institute of Technology, 1990

[9] Roberto Amadini
*Definizione e trattamento del vincolo di cardinalità insiemistica nella libreria JSetL*
Tesi di laurea triennale, Università di Parma, 2007

[10] Daniele Pandini
*Progettazione e realizzazione in Java di un risolutore di vincoli su domini finiti*
Tesi di laurea triennale, Università di Parma, 2007

[11] Gilad Bracha
*Generics in the Java Programming Language*
Sun Microsystems, 2004