Indice

Capitolo 1

Introduzione

La programmazione con vincoli costituisce una tecnica relativamente nuova di programmazione che presenta diversi aspetti interessanti sia dal punto di vista teorico, che metodologico, che applicativo.

I linguaggi di programmazione con vincoli sono, in particolare, elemento essenziale della così detta "programmazione dichiarativa"in cui l'attenzione del programmatore viene posta soprattutto sul "cosa" si deve risolvere piuttosto che sul "come" risolvere il problema.

Nel passato sono stati sviluppati diversi nuovi linguaggi di programmazione con vincoli, in particolare nell'ambito dei linguaggi logici: CHIP [9], clp(FD) [7], CLP(BNL) [14], CLP($\mathcal{SECT}$) [8] (dove CLP sta per Constraint Logic Programming).

I linguaggi logici si prestano particolarmente bene ad "ospitare" la nozione di vincolo e i meccanismi per la loro gestione, offrendo già elementi essenziali per questi scopi quali la nozione di variabile "logica". I linguaggi logici con vincoli offrono però modalità di programmazione piuttosto distanti da quelle della programmazione tradizionale a cui la maggior parte dei programmatori è abituata. Un approccio alternativo consiste perciò nel cercare di estendere i linguaggi di programmazione convenzionali, quale il C++ o Java, per includere costrutti e meccanismi che permettano di supportare la programmazione con vincoli, e più in generale una programmazione dichiarativa, pur rimanendo all'interno di un contesto di programmazione più convenzionale.

Per fare ciò si possono considerare due diverse alternative:

1. Estensione del linguaggio.

2. Definizione di una libreria.

Alcuni esempi di estensioni di linguaggi sono dati da: Alma-0 [2, 3], SINGLETON [16] e DJ [18, 19]. Alma-0 è un linguaggio di programmazione che estende la programmazione imperativa con un numero limitato di costrutti ispirati dal paradigma di programmazione logico. SINGLETON è un linguaggio dichiarativo, basato sulla nozione di insieme, che combina le caratteristiche dei linguaggi convenzionali con alcune caratteristiche dei linguaggi CLP, come il non determinismo, l'unificazione, la risoluzione dei vincoli. DJ (Declarative Java) è un estensione del linguaggio Java che supporta la programmazione con vincoli.

Le librerie JSolver [5] e ILOG SOLVER [11, 15] offrono invece un esempio del secondo tipo di approccio. JSolver è una libreria Java e ILOG SOLVER una libreria C++. Entrambe supportano la programmazione con vincoli su domini finiti.

I linguaggi orientati agli oggetti come il C++ o Java offrono importanti vantaggi nella definizione di una libreria. Infatti permettono di aggiungere nuove possibilità al linguaggio (nuovi tipi, nuove operazioni...) mantenendo una buona separazione tra interfaccia ed implementazione.

In questo lavoro di tesi proponiamo l'implementazione di una libreria Java, che abbiamo chiamato JavaSet, con la quale cerchiamo di estendere il paradigma della programmazione orientata agli oggetti con alcune caratteristiche fondamentali dei linguaggi CLP: l'utilizzo di variabili logiche, l'unificazione, il non determinismo, la risoluzione dei vincoli.

I vincoli che vogliamo introdurre e risolvere sono vincoli insiemistici: uguaglianza tra variabili logiche e in particolare tra insiemi e tra liste, appartenenza di un elemento ad un insieme, unione, disgiunzione e altre operazioni insiemistiche di base (intersezione, inclusione ...).

Gli algoritmi di risoluzione che adottiamo sono quelli di $SAT_{s\varepsilon\tau}$ cioè del risolutore di vincoli del linguaggio CLP($\mathcal{SET}$) [8].

La dissertazione è organizzata come segue.

Capitolo 2 In questo capitolo vengono descritte alcune proposte per la programmazione con vincoli in Java e C++: JSolver, DJ, ILOG SOLVER.

Capitolo 3 Da questo capitolo inizia la descrizione di JavaSet. Qui vengono presentate le caratteristiche principali delle variabili logiche, delle liste e degli insiemi e le operazioni definite su di essi.

Capitolo 4 In questo capitolo vengono introdotti i concetti di vincolo e di risoluzione dei vincoli e vengono descritti i vincoli definiti in JavaSet.

Capitolo 5 In questo capitolo viene descritta l'implementazione delle classi più importanti della libreria e la realizzazione in JavaSet di alcuni costrutti tipici nei linguaggi CLP: le variabili logiche, la risoluzione dei vincoli, il non determinismo.

Capitolo 6 In questo capitolo vengono descritti alcuni esempi che mostrano in che modo si può utilizzare la libreria per risolvere problemi di soddisfacimento di vincoli.

Capitolo 7 In questo capitolo vengono descritte le principali differenze tra Java e C++ per chiarire meglio alcuni aspetti implementativi della realizzazione della libreria.

Capitolo 8 Questo capitolo conclude la descrizione della libreria e fa un accenno ai lavori futuri.

Capitolo 2

Programmazione con vincoli in Java e C++

Negli ultimi anni la programmazione con vincoli ha riscosso un notevole interesse; molti problemi reali possono essere modellati come problemi di soddisfacimento di vincoli (CSP) e quindi risolti con l'uso delle tecniche della programmazione con vincoli. In passato sono stati implementati molti linguaggi, ad esempio CLP(fd) [7](CLP sta per Constraint Logic Programming) o CHIP [9]. Questi linguaggi sono stati ottenuti estendendo l'unificazione del Prolog con i risolutori di vincoli.

I linguaggi logici, come ad esempio il Prolog si prestano particolarmente bene ad "ospitare" la nozione di vincolo e i meccanismi per la loro gestione, offrendo già elementi essenziali per questi scopi quali la nozione di variabile "logica". Tuttavia ci sono anche delle limitazioni sia in termini di integrazioni con altri software scritti in altri linguaggi sia in termini di estendibilità. Una possibile alternativa consiste perciò nel cercare di integrare i concetti dei linguaggi CLP in linguaggi di programmazione convenzionali, come C++ o Java, senza rinunciare alla dichiaratività e all'efficienza.

In questo capitolo descriveremo alcune proposte di programmazione con vincoli in Java e C++: JSolver, DJ, ILOG SOLVER.

2.1 JSolver

JSolver [5] è una libreria Java che estende la struttura orientata agli oggetti di Java con la programmazione dichiarativa basata sui vincoli. L'implementazione è stata realizzata, come per JavaSet, con un set di pure classi Java. JSolver supporta la programmazione con vincoli su variabili intere e booleane e permette di implementare la parte logica della programmazione con vincoli direttamente in Java, senza nessuna chiamata primitiva al C++ e CLP; JSolver infatti può essere inserita in un server per eseguire le operazioni run-time. Questa è una diversa filosofia di progettazione se confrontata con DJ [18] (cfr. sezione 2.2 pag. 13). Il codice DJ è compilato usando il B-Prolog, quindi dato un problema con vincoli è il B-Prolog a trovarne la soluzione. DJ è adatto alla risoluzione di problemi statici, mentre non è adatto ad un sistema in cui sono richieste continue risoluzioni dovute a variazioni nella definizione del problema. JSolver si presta meglio alla risoluzioni di problemi dinamici che implichino l'interazione con l'utente ed eventi run-time. In DJ è il B-Prolog che fornisce il meccanismo di risoluzione dei vincoli, mentre in JSolver questo meccanismo è realizzato utilizzando Java, come avviene anche in JavaSet.

2.1.1 Classi principali di JSolver

La maggior parte delle possibilità di JSolver sono accessibili dalla classe d'utilità `JSolver` e la maggior parte dei vincoli possono essere realizzati mediante metodi forniti dalle classi `Var` e `VarVector`. Come si vede dall'esempio che segue lo stile di programmazione è tipicamente dichiarativo:

Esempio 1

```
public class Exemple {
  public static void main(String argv[])throws FailException {
    Var a =JSolver.var(0, 20, "a");
    Var b = JSolver.var(0, 20, "b");
    JSolver.post(a.diff(b).gt(10));
```

```
        VarVector vars = JSolver.varVector(a,b);
        JSolver.solve(JSolver.generate(vars));
        System.out.println(vars);
      }
    }
```

In questo esempio vengono create due variabili intere vincolate, entrambe con un dominio da `0` a `20` e con un nome, sulle quali viene posto il vincolo $a - b > 10$ mediante l'istruzione `JSolver.post(a.diff(b).gt(10))`, che è del tutto simile a quella che useremmo in JavaSet per introdurre lo stesso vincolo nel constraint store. Le variabili sono poi inserite in un vettore che viene passato come parametro al metodo `generate()` della classe `JSolver`, questo metodo istanzia le due variabili e quindi crea un goal per `JSolver.solve()`. Il metodo `JSolver.solve()` realizza la ricerca non deterministica.

2.1.2 Variabili JSolver

Come si può vedere nell'Esempio 1 in JSolver le variabili intere vincolate sono create semplicemente fornendo i due estremi del dominio e un parametro opzionale per il nome.

```
Var x = JSolver.var(0,10,"x"); // variabile intera x[0..10]
```

Le variabili booleane invece hanno soltanto il parametro opzionale per il nome.

```
Var y = JSolver.boolVar("y"); // variabile booleana y[true,false]
```

Le variabili così costruite possono essere memorizzate in un vettore usando la classe `VarVector` che ha la stessa interfaccia della classe Vector di Java con l'aggiunta dei metodi per la definizione dei vincoli. I vettori vengono usati per semplificare il passaggio degli argomenti delle variabili. Inoltre ci sono vincoli che possono essere applicati ad un intero vettore, come ad esempio il vincolo di somma.

I vettori possono essere creati in diversi modi:

```
JSolver.varVector(x, y);
JSolver.varVector(2, 0, 10); // due variabili con dominio [0..10]
JSolver.varVector();         // vettore non inizializzato
```

2.1.3 Vincoli JSolver

Dopo aver creato le variabili e i vettori, su di essi si possono definire i vincoli. JSolver memorizza i vincoli in un "contenitore" che fa sì che la propagazione si realizzi efficientemente senza ulteriori ricerche run-time.

JSolver fornisce un set di vincoli unari e binari; quelli che seguono, ad esempio, sono vincoli di disuguaglianza sulle variabili `x` e `y`:

```
JSolver.post(x.neq(5))
JSolver.post(y.neq(x))
```

Inoltre in JSolver è possibile imporre i seguenti vincoli sui vettori: il *vincolo "all different"*, i *vincoli di cardinalità* e i *vincoli di somma*. Il *vincolo "all different"* impone che gli elementi del vettore siano tutti diversi tra loro; i *vincoli di cardinalità* servono per stabilire relazioni di cardinalità per gli elementi del vettore, ad esempio possono stabilire che la cardinalità sia maggiore, minore o uguale ad un dato valore; infine *i vincoli di somma* definiscono un vincolo per il valore totale delle variabili del vettore; per esempio la somma di tutte le variabili potrebbe essere vincolata ad essere inferiore ad un dato valore. L'utente inoltre può facilmente creare altre classi di vincoli semplicemente estendendo la classe `Constraint` di JSolver e sovraccaricandone i pochi metodi astratti.

2.1.4 Classi reversibili

Tutte le variabili vincolate e i vincoli di JSolver sono "reversibili", cioè durante il backtracking possono tornare ad uno stato precedente. JSolver permette di applicare la reversibilità anche alle variabili definite dall'utente e a quelle non vincolate, anche

se questa necessità si presenta raramente per tali variabili. JSolver prevede un set di classi per rappresentare i valori che potrebbe essere necessario riconvertire a valori precedenti in conseguenza del backtracking. Ad esempio sono previste le classi `RevInteger`, `RevFloat` e `RevBoolean` rispettivamente per gli interi, i reali e i booleani reversibili; queste classi sono analoghe alle corrispondenti classi wrapper del Java, con in più la capacità di annullare i valori assegnati in caso di backtracking.

2.1.5 Risoluzione dei vincoli in JSolver

Dopo che le variabili vincolate sono state create e sono stati definiti i vincoli su di esse, inizia la ricerca della soluzione. Come succede anche in JavaSet, se necessario, durante la ricerca, verranno creati dinamicamente altri vincoli ed altre variabili.

L'algoritmo di ricerca di JSolver seleziona una variabile per volta da un dato insieme di variabili da istanziare. Per default, le variabili sono selezionate nell'ordine in cui sono state memorizzate. JSolver fornisce un set di strategie predefinite, tuttavia è possibile implementarne altre estendendo la classe JSolver `ChooseVarHeuristic`. Quando tutte le variabili sono istanziate, allora si è trovata una soluzione. Per assegnare un valore ad una variabile si procede scegliendo sempre come primo valore il valore minimo del dominio e poi eventualmente prendendo nell'ordine i successivi. Ancora un volta JSolver fornisce un set di strategie, ma se ne possono implementare altre estendendo la classe Jsolver `SelectValueHeuristic`.

Dopo l'assegnamento di un valore ad una variabile, avviene la propagazione dei vincoli che può eventualmente ridurre i domini delle variabili soggette a qualche vincolo. Se qualche dominio diventa nullo si verifica un fallimento. In questo caso viene testato un altro valore; se nessun altro valore è disponibile, l'algoritmo di JSolver torna, con la procedura di backtracking, ad un punto di scelta precedente che abbia delle alternative aperte, e continua la ricerca da quel punto. Se non viene trovato nessun altro punto di alternativa, il programma solleva un eccezione di tipo `FailException`.

2.2 DJ

DJ [18] (Declarative Java), a differenza di JavaSet e di JSolver, non è una libreria Java, ma una vera e propria estensione sintattica del linguaggio che amalgama Java con la programmazione con vincoli. DJ semplifica notevolmente la costruzione di interfacce grafiche (GUI) e applets. L'utente deve solo specificare i componenti che costituiscono l'interfaccia grafica e introdurre le relazioni tra i vari componenti usando i vincoli. La configurazione per i componenti è poi determinata automaticamente dal sistema. Inoltre DJ, come linguaggio di programmazione con vincoli, ha il vantaggio di esprimere i problemi e le loro soluzioni nello stesso linguaggio e poiché DJ usa Java come linguaggio oggetto, i risultati possono essere distribuiti sul World Wide Web come applets Java o inclusi in altre applicazioni più estese.

Il compilatore per DJ è implementato in B-Prolog [17], un sistema di programmazione logica con vincoli. Per un programma DJ, il compilatore prima estrae un problema di soddisfacimento di vincolo dal programma, quindi invoca il risolutore di vincoli per risolvere il problema e determinare i valori per i componenti, infine genera un programma Java e un file HTML dal programma originale DJ e la soluzione ottenuta dal risolutore.

2.2.1 Struttura di un programma DJ

DJ supporta la programmazione con vincoli e mantiene la struttura orientata agli oggetti di Java. Un programma DJ consiste di varie classi e eventualmente varie definizioni di vincolo (cf. [19]). Oltre alle definizioni di campi e metodi, una classe può anche includere *dichiarazioni di campi-dj*, *azioni* e *vincoli*.

```
DichiarazioneClasseMembro::−
  DichiarazioneCampi
  DichiarazioneMetodi
  DichiarazioneCampiDJ
  Vincoli
  Azioni
```

Le *dichiarazioni di campi-dj* servono per dichiarare i componenti grafici delle classi e gli attributi delle classi. Ogni campo-dj è il nome di un particolare oggetto che è creato automaticamente quando viene creata un'istanza della classe che lo contiene. Diversamente dalle variabili Java, che possono essere aggiornate, i campi-dj sono variabili i cui valori sono determinati automaticamente dal sistema basandosi sui vincoli definiti su di esse. Un'*azione*, associata con un'istanza, specifica l'azione da eseguire quando una certa condizione si verifica su quell'istanza, la sintassi per la descrizione di un'azione è la seguente:

```
command(Component,MethodInvocation)
```

Infine i *vincoli* sono relazioni tra le istanze delle classe predefinite o definite dall'utente; un vincolo ha una sintassi simile a quella di un'espressione condizionale in Java ed una definizione di vincolo è come la definizione di un metodo ma il suo corpo è una sequenza di vincoli. Grazie a questa somiglianza un utente, che conosca Java e la programmazione con vincoli, può imparare a programmare con DJ molto velocemente senza rischiare di confondere gli statements con i vincoli, in quanto compaiono in contesti diversi.

Vediamo un semplice esempio per la costruzione di un applet che genera un `Hello World` button:

```
class HelloWorld {
  dj Button bt{text == "Hello World!"};
}
```

Il programma è costituito dalla sola classe `HelloWorld`, la linea che inizia con `dj` è una *dichiarazione di campo-dj*. Essa dichiara un componente di nome `bt` che è un'istanza della classe `Button`. L'espressione tra parentesi graffe dopo `bt` è un *vincolo* che appunto vincola l'attributo `text` di `bt` ad essere ″`Hello World!`″.

2.2.2 Classi base e tipi

La creazione dei componenti può avvenire anche usando le *classi base*. Ogni classe base ha degli attributi e ogni attributo ha un tipo, un nome ed eventualmente un vincolo che restringe il dominio dei valori dell'attributo.
Ad esempio consideriamo le seguenti dichiarazioni di attributi:

```
dj int a in 0..500;
dj int b == a;
```

La prima linea dichiara un intero `a` e vincola il suo dominio ad essere il set degli interi tra `0` e `500`, inclusi. La seconda linea dichiara un intero `b` e lo vincola ad essere uguale ad `a`. I possibili tipi per gli attributi sono i seguenti: `int`, `boolean`, `char`, `String`, `Color`, `Font`, `Dimension` e `Point`. Tutte le classi base sono sottoclassi della classe `DJComponent`.

Ovviamente oltre ad utilizzare le classi base, l'utente può definire nuove classi. La dichiarazione dei campi della nuova classe sarà di questo tipo:

```
dj Type DichiaratoreCampoDj,..., DichiaratoreCampoDj;
```

Dove `Type` dovrà essere uno dei tipi visti in precedenza, una classe base, o una classe DJ definita dall'utente, e ogni `dj Type DichiaratoreCampoDj` dovrà essere un identificatore o un identificatore su cui è specificato qualche vincolo, ad esempio:

```
dj int a;
dj int b == a;
dj Color color in{red,black,white};
dj int c in 0..9;
dj Rectangle rect1, rect2{size == rect1.size};
```

Nel definire nuove classi DJ, l'utente può inoltre sfruttare la struttura orientata agli oggetti di Java, una classe può estendere un'altra classe, ad esempio sfruttando l'ereditarietà possiamo definire la seguente classe `HelloWorldButton`:

```
class HelloWorldButton extends Button {
      text == "Hello World!";
}
```

In questo modo la classe `HelloWorldButton` eredita tutti gli attributi da `Button`.

2.2.3 Vincoli DJ

Come si è accennato in precedenza, sui campi delle classi possono essere definiti diversi tipi di vincolo i quali possono comparire come membri delle dichiarazioni o separati da essi. Consideriamo di nuovo l'esempio `HelloWorld`, possiamo scrivere il vincolo sul componente `bt` anche separatamente dalla sua dichiarazione:

```
class HelloWorld {
      dj Button bt;
      bt.text == "Hello World!";   // constraint
}
```

I vincoli in DJ si dividono in *vincoli base* e *vincoli composti*. I vincoli base sono: di *dominio*, *aritmetici*, *simbolici*, o una chiamata ad un vincolo definito da utente, un vincolo composto è dato da vincoli base e connettivi.
Un vincolo di dominio è del tipo:

```
A in D
```

dove A è una variabile e D un dominio cioè un intervallo di interi o una lista di valori nella forma $\{a_1, a_2, ..., a_n\}$. Il tipo di A e D deve essere lo stesso.
Un vincolo aritmetico è della forma:

```
expression R expression
```

dove `expression` è un'espressione lineare costituita da vincoli su interi, da variabili intere, dalle funzioni `sum`, `minimum` and `maximum` e dai quattro operatori aritmetici $+$, $-$,$*$, / ; e R è uno dei seguenti simboli relazionali: $=$, != , $>=$, $>$, $<=$, $<$.

Un vincolo simbolico permette di definire delle relazioni tra oggetti senza riferirsi direttamente ai loro attributi.

Sono ammessi i seguenti vincoli:

- **Vincoli di uguaglianza e disuguaglianza**. `A == B` e `A != B`, con `A` e `B` che possono essere delle costanti o delle variabili ma devono essere dello stesso tipo.
- **Vincoli di posizione.** Questi vincoli mettono in relazione le posizioni di due componenti.
- **Vincoli** `same...(O1,O2)`. Questi vincoli forzano O1 e O2 ad avere lo stesso valore per l'attributo specificato.

I vincoli base possono poi essere combinati fra loro con l'uso dei connettivi per formare i vincoli composti.

In Dj sono ammessi i seguenti vincoli composti:

- $\sim C$
- $C1 \parallel C2$
- C1 && C2
- $C1 \rightarrow C2$

Oltre ad utilizzare questi vincoli l'utente ha poi la possibilità di definirne dei nuovi. La dichiarazione avviene nella seguente forma:

```
constraint Name(FormalParameterlist constraintBody)
```

che definisce un vincolo di nome `Name` fra una lista di parametri formali (nella quale cioè per ogni parametro si indica tipo e nome), e `constraintBody` è il blocco dei vincoli.

Le definizioni di vincolo non risiedono in nessuna classe e possono essere utilizzate da tutte le classi. Questo aspetto è fonte di dibattito in quanto una struttura di

questo tipo non è conforme alla struttura orientata agli oggetti di Java. D'altra parte mettere le definizioni di vincolo in una delle classi coinvolte renderebbe la descrizione innaturale.

2.2.4 Array e vincoli su array

In DJ è anche possibile definire array di variabili vincolate e definire dei vincoli su di essi o sulle variabili che li compongono. Consideriamo ad esempio di dover definire un pannello che consiste di diversi bottoni ciascuno dei quali mostra il messaggio "Hello" in una diversa lingua:

```
class Hellos {
      component Button hellos[10] {
            {text == "Hello";},
            {text == "Bonjour";},
            {text == "Buongiorno";},
            {text == "Kon Ni Chi Wa";},
            ...
            {text == "Ni Hao";}
      }
}
```

Questa classe ha dieci bottoni ciascuno dei quali contiene un testo diverso.

La dichiarazione di un array avviene in una delle seguenti forme:

```
Identifier DimExprs
Identifier DimExprs in D
Identifier DimExprs R Expr
Identifier DimExprs ConstraintBlock
```

Dove la prima dichiara un array e non impone nessun vincolo su di esso; la seconda dichiara che su ogni elemento `X` dell'array viene posto il vincolo `X in D`; la terza dichiara che su ogni elemento `X` dell'array viene posto il vincolo `X R Expr`; la quarta impone i vincoli sugli elementi dell'array come segue: Sia A l'identificatore

per l'array in questione, se il `ConstraintBlock` è costituito da un solo blocco di vincoli, allora i vincoli sono applicati a tutti gli elementi dell'array. Viceversa se il `ConstraintBlock` è costituito da una sequenza di blocchi di vincoli $\{B_0, \ldots, B_{n-1}\}$ (in questo caso la dimensione dell'array deve essere uguale a n) allora ricorsivamente i vincoli del blocco B_i vengono applicati a `A[i]` per $i = 0, ..., n-1$.

Gli array sono utili anche per imporre vincoli su di un gruppo di componenti; a volte il gruppo di componenti è di tipo arbitrario, in questo caso DJ prevede l'utilizzo di *arrays anonimi*. Un array anonimo è un array senza nome che può essere costruito in due modi. Un modo è quello di elencare i suoi elementi esplicitamente come segue:

```
{A1,...,An}
```

dove gli elementi possono essere di tipi diversi.

L'altro modo per la costruzione di array anonimi è quello di usare un set di espressioni come segue:

```
array(Exp,Enumerator,...,Enumerator)
```

dove `Exp` è un'espressione e `Enumerator` è un vincolo di dominio nella forma `v in D` o un vincolo relazionale. Sia `Vars` un set di variabili che appare in tutti gli enumeratori, per ogni tipo di valore di `Vars` che soddisfa gli enumeratori, `Exp` ha un valore. Quest'espressione rappresenta un array di tutti tali valori.

A volte è necessario imporre vincoli su ogni elemento di un array. DJ per questo scopo fornisce il vincolo `for` in questa forma:

```
for (Enumerator,...,Enumerator) Constraint;
```

dove `Enumerator` è un vincolo di enumerazione. Sia `Vars` un set di variabili che appare in tutti gli `Enumerator`, per ogni tipo di valore di `Vars` che soddisfa gli enumeratori, `Constraint` deve essere soddisfatto.

Un altro vincolo su array fornito da DJ è il vincolo `Alldifferent`; dato un array di elementi di un certo tipo (int, char, String, Color o Font) assicura che tutti gli elementi siano diversi tra loro.

2.2.5 Risoluzione con DJ del problema delle N-regine

La possibilità di introdurre questi vincoli su arrays rende DJ un linguaggio adatto non solo alla costruzione di grafici e interfacce grafiche, ma anche alla risoluzione di problemi più generali di soddisfazione di vincoli. Vediamo appunto un esempio che chiarisce questo aspetto: il problema delle N-regine.

Il problema è descritto come segue: Su una scacchiera N×N si richiede di posizionare N regine in modo tale che non si attacchino a vicenda, i.e., due regine non possono trovarsi sulla stessa riga, sulla stessa colonna o sulla stessa diagonale.

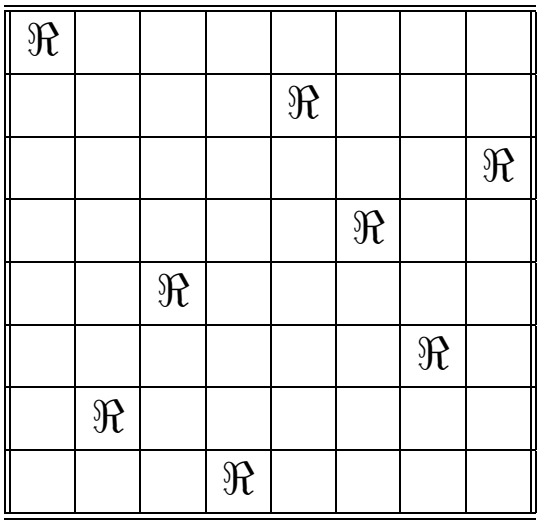


Figura 2.1: Una soluzione del problema delle 8-regine

In figura 2.1 si può vedere una possibile soluzione del problema delle 8-regine. Il codice DJ per la risoluzione del problema è il seguente:

```
class Queens {
    static public final int N = 8;
    dj Square board[N][N]{fill == false};
    dj Image queens[N]{name == "queen.gif";
                       size == board[0][0].size};
    dj int Pos[N] in 0..N-1;
```

```
    for (i in 0..N-1,j in 0..N-1){
         pos[i]==j -> samePosition(queens[i],board[i][j]);
    }
    notattack(pos,N);
    grid(board)
}

constraint notattack(int[] pos,int N){
   for (i in 0..N-2,j in i+1..N-1){
        pos[i] != pos[j];
        pos[i] != pos[j]+j-i;
        pos[i] != pos[j]+i-j;
   }
}
```

`N` è definito come una costante. La scacchiera è rappresentata come un array bidimensionale di caselle quadrate vuote e le `N` regine sono rappresentate come un array di immagini. Ogni immagine ha la stessa dimensione dei quadrati della scacchiera. L'array `pos` serve ad indicare le posizioni delle regine. Per ogni regina `i`, l'elemento `pos[i]` indica il numero della riga in cui è piazzata la regina.

Il vincolo `for` ha questo significato: Per ogni regina `i` e ogni riga `j` se la regina `i` è piazzata nella riga `j`, allora l'immagine `queen[i]` e la casella `board[i][j]` devono avere la stessa posizione. Il vincolo:

```
pos[i]==j → samePosition(queens[i],board[i][j]);
```

è detto *vincolo di implicazione*. Un vincolo di implicazione `A` → `B` significa che il vincolo `B` deve essere soddisfatto se lo è il vincolo `A`. Il vincolo `notattack` è definito dall'utente e impone le condizioni necessarie affinchè due regine non si attacchino a vicenda. Il vincolo `grid` è necessario per il layout della scacchiera; assicura che gli elementi dell' array `board` siano piazzati su una griglia.

2.3 Ilog Solver

ILOG SOLVER [10, 11, 15] è una libreria C++ per la programmazione con vincoli con domini finiti. In ILOG SOLVER i concetti della programmazione logica con vincoli sono implementati utilizzando le classi: le variabili logiche, i vincoli, e gli algoritmi di ricerca (*g*oals) sono oggetti.

Tutte le variabili logiche e i vincoli sono memorizzati nello *s*tore. Quando una variabile logica non ha un valore, le viene associato un dominio; questo dominio rappresenta, implicitamente o esplicitamente, i valori che la variabile può assumere.

In ILOG SOLVER un'istruzione consiste nel creare una nuova variabile oppure nell'aggiungere un vincolo allo store. Nel secondo caso viene attivato l'algoritmo di propagazione dei vincoli detto "*constraint propagation*" che elimina dai domini delle variabili logiche i valori inconsistenti, in questo modo può accadere, che il sistema venga risolto, che si trovi un'inconsistenza, oppure nessuna delle due cose.

L'algoritmo di propagazione è incompleto, cioè non trova tutte le inconsistenze. Questo perchè si è voluto un costo di esecuzione polinomiale e quindi è stato necessario rinunciare alla completezza. Per ovviare all'incompletezza, i linguaggi CLP si affidano alla ricerca non deterministica: si considera un punto di scelta e si applica l'algoritmo di propagazione. Se si arriva ad un'inconsistenza si applica il backtracking, cioè si torna al punto di scelta più vicino, si annulla l'ultimo vincolo introdotto, insieme alle sue conseguenze, e si tenta un'altra alternativa.

2.3.1 Variabili logiche di ILOG SOLVER

In ILOG SOLVER una variabile logica è un oggetto C++ con diversi campi nascosti, che comprendono il valore della variabile, se è noto, la lista dei vincoli posti su di essa, e una rappresentazione del dominio, se il valore non è noto. Ogni tipo di variabile è implementato da una classe. Ad esempio, la classe per le variabili intere (`CtIntVar`) è diversa dalla classe delle variabili di tipo reale (`CtFloatVar`). ILOG SOLVER prevede varie classi di variabili: le variabili intere, il cui dominio è un

intervallo o un'enumerazione di interi; variabili reali, il cui dominio è un intervallo; le variabili booleane, il cui dominio è l'insieme {0,1}; gli insiemi.

Si possono usare diversi tipi di variabile nella stessa applicazione, in quanto l'algoritmo di propagazione tratta in modo uniforme tutte le variabili.

2.3.2 Gestione della memoria

In ILOG SOLVER le variabili sono oggetti e come tali possono essere allocate e create come qualunque altro oggetto C++. Si usa l'operatore `new` per allocarle e un costruttore per inizializzarle. La gestione degli oggetti allocati nell'heap è lasciato all' utente, che è responsabile della deallocazione esplicita della memoria quando un oggetto non è più utile. Per la deallocazione si utilizza l'operatore `delate`, come è usuale in C++.

Tuttavia, poiché nell'uso più comune di ILOG SOLVER si utilizza una ricerca non deterministica che corrisponde ad una ricerca in profondità, ILOG SOLVER fornisce un allocatore il quale, quando un ramo viene abbandonato, dealloca automaticamente la memoria utilizzata dagli oggetti creati in quel ramo dell'albero.

Ad esempio l'istruzione seguente crea una variabile intera utilizzando la gestione della memoria di ILOG SOLVER:

```
CtIntVar* z = new (CtHeap()) CtIntVar(-10,10);
```

2.3.3 I Vincoli di ILOG SOLVER

In ILOG SOLVER un vincolo è un oggetto C++. L'introduzione di un vincolo consiste nel creare tale oggetto e nell'aggiungerlo alla lista dei vincoli delle variabili appropriate.

Ad esempio, l'istruzione che segue introduce un vincolo:

```
CtTell(x == (y + z));
```

In ILOG SOLVER sono definiti i seguenti vincoli base: $=$, $\leq$, $\geq$, $<$, $>$, $+$, $-$, $*$, $/$, inclusione, unione, intersezione, appartenenza, or booleano, and booleano, not booleano, xor booleano. Inoltre sono disponibili alcune versioni generali di questi vincoli. Ad esempio `CtAllNeq` impone il vincolo di disuguaglianza tra tutti gli elementi di un array, oppure il vincolo `CtArraySum` restituisce una variabile uguale alla somma delle variabili in un dato array. Questi vincoli sono molto utili perchè permettono di allocare un solo vincolo qualunque sia il numero di variabili.

ILOG SOLVER permette anche di creare nuovi vincoli componendo i vincoli predefiniti usando gli operatori lgici *or*, *and*, *xor* e *not*. Questi operatori sono implementati utilizzando gli operatori booleani del C++: $\|$, &&, ! =, !. Inoltre ILOG SOLVER offre due funzioni primitive per l'introduzione dei vincoli: `CtTell` che introduce una formula booleana e `CtIfThen` che introduce una implicazione logica tra due formule booleane.

Ad esempio l'istruzione seguente indica che almeno una delle due variabili `x` e `y` deve essere uguale a `0`:

```
CtTell((x == 0) || (y == 0));
```

Quando la composizione non è sufficiente, è possibile definire una nuova classe di vincoli, i cui metodi descrivono come il vincolo deve essere propagato.

2.3.4 Ricerca della soluzione

Un'altra caratteristica dei linguaggi CLP è la capacià di sviluppare algoritmi in grado di scegliere tra diverse possibilità senza sapere a priori qual è quella esatta. Il C++ non prevede questa possibilità.

Anche in questo caso ILOG SOLVER ricorre agli oggetti. Un algoritmo non deterministico è definito con degli oggetti detti *goals*. Un goal è un oggetto con un particolare metodo che definisce in che modo deve essere eseguito. Un goal fallisce se durante la sua esecuzione viene trovata un'inconsistenza.

Utilizzando la funzione `CtAnd` è anche possibile creare un goal che è una sequenza

di altri goal e utilizzando la funzione `CtOr` si può crea un goal scegliendo tra altri goal. L'esecuzione di questo goal crea un punto di scelta, quindi inizia ad eseguire i goals fino a che uno non ha successo.

Al fine di semplificare la sintassi, per definire una classe goal viene utilizzata la macro `CTGOALn`, con un nome e la lista dei parametri come argomenti, `n` è il numero dei parametri. Questa macro è seguita dal corpo del metodo che esegue il goal.

Ad esempio, il goal seguente sceglie un valore per la variabile. Per prima cosa seleziona un valore, poi sceglie se assegnare il valore alla variabile o rimuovere il valore dal dominio della variabile, quindi fa una chiamata ricorsiva a se stesso.

```
CTGOAL1(CtInstantiate, CtIntVar*,x){
   CtInt a = x -> chooseValue();
   CtOr(Constraint(x == a),
        CtAnd(Constraint(x != a),
              CtInstantiate(x)));
}
```

In Prolog invece i punti di scelta sono impliciti nel linguaggio, infatti basta definire due o più clausole con lo stesso nome. Il codice Prolog equivalente a quello appena visto è il seguente.

```
Instantiate(x):- ChoseValue(x,a), InstantiateAux(x,a).

InstantiateAux(x, a):- Constraint(x = a).
InstantiateAux(x, a):- Constraint(x != a), Instantiate(x).
```

Questo esempio mostra una delle caratteristiche più importanti di ILOG SOLVER, cioè che l'utente non si deve occupare del non determinismo quando non è necessario. Ad esempio, la funzione `chooseValue` è implementata direttamente come una funzione C++.

ILOG SOLVER offre altre due funzioni di controllo: `CtSolve` e `CtMinimize`. La prima ricerca in modo non deterministico una soluzione per il goal, la seconda ricerca una soluzione che minimizzi il valore della variabile logica data. L'algoritmo

utilizzato è simile a quello descritto in [9]. E' anche possibile etichettare un punto di scelta e poi, durante il backtracking, tornare a quel punto etichettato.

Durante il backtracking vengono annullati gli effetti della propagazione dell'ultimo vincolo risolto. Per realizzare questo, è stato utilizzato un meccanismo che tiene traccia dei cambiamenti occorsi alle variabili e al constraint store in modo da poterli annullare, se necessario. Inoltre è anche possibile salvare lo stato di un qualunque oggetto C++ in modo che venga ripristinato dopo il backtracking.

Capitolo 3

Variabili logiche e strutture dati in JavaSet

In JavaSet abbiamo previsto l'esistenza di variabili logiche, insiemi e liste. La libreria fornisce la classe `Lvar`, per la creazione delle variabili logiche, la classe `Lst` per le liste, e la classe `Set` per gli insiemi. In questo capitolo presentiamo le caratteristiche principali di questi nuovi oggetti e le operazioni definite su di essi.

3.1 Variabili logiche: la classe Lvar

I linguaggi (logici) per la programmazione con vincoli prevedono dei costrutti che non sono tipicamente presenti nei linguaggi di programmazione imperativi o in quelli orientati agli oggetti. In particolare Java non prevede la possibilità di utilizzare una variabile prima che questa sia stata inizializzata: se si tenta di farlo si incorre in un errore compile-time. Al contrario, i linguaggi logici come ad esempio il Prolog, ma anche alcuni linguaggi imperativi per la programmazione con vincoli (cfr. Alma-0 [2, 3]), permettono di utilizzare variabili logiche: queste variabili non devono necessariamente essere inizializzate al momento della loro dichiarazione, restano non inizializzate fino a quando non viene loro assegnato un valore. L'accesso a queste variabili è possibile anche quando non hanno alcun valore e non genera nessun errore. Inoltre, contrariamente a quanto accade per le variabili dei linguaggi

imperativi, dopo che ad una variabile logica è stato assegnato un valore, questo non può più essere modificato da un altro assegnamento.

Nella definizione della nostra libreria, abbiamo previsto l'esistenza di oggetti che avessero le stesse caratteristiche di una variabile logica. Tali oggetti sono realizzati come istanze della classe `Lvar`. Una `Lvar` può essere inizializzata al momento della creazione oppure può essere dichiarata come variabile non inizializzata e assumere, eventualmente, un valore nel corso della computazione, in conseguenza dei vincoli posti su di essa.

Consideriamo, ad esempio le seguenti definizioni di variabili `Lvar`:

```
Lvar X = new Lvar();     // X: Lvar non inizializzata
Lvar y = new Lvar(2);    // y: Lvar inizializzata con intero
```

Le variabili `Lvar` sono oggetti, e come tali vengono allocati con l'operatore `new` e inizializzati da un costruttore, come è usuale in Java. Nel nostro esempio `X` è una `Lvar` non inizializzata che nel corso della computazione potrà diventare inizializzata oppure no, `y` invece è inizializzata, il suo valore è `2` e non potrà più essere modificato.

Il valore memorizzato in una `Lvar` può essere di tipo primitivo (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`), un oggetto di una classe wrapper (`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`), o un oggetto `String`; inoltre una `Lvar` può contenere un oggetto della classe `Lst`, o della classe `Set` come nel caso delle `Lvar` che seguono:

```
Lst l = new Lst();        // l: lista non inizializzata
Lvar A1 = new Lvar(l);    // A1: Lvar inizializzata con lista
Lvar A2 = new Lvar(new Lst());
                          // A2: Lvar inizializzata con lista
Set s = new Set();        // s: insieme non inizializzato
Lvar B1 = new Lvar(s);    // B1: Lvar inizializzata con insieme
Lvar B2 = new Lvar(new Set());
                          // B2: Lvar inizializzata con insieme
```

Si noti che `A1` e `A2` sono entrambe `Lvar` inizializzate contenenti una lista non inizializzata.

Una variabile logica contenente una lista può essere creata passando al costruttore un riferimento ad una `Lst` creata precedentemente, come nel caso di `A1`, oppure creando la `Lst` quando si crea la `Lvar`, come nel caso di `A2`. La stessa cosa vale per le `Lvar` contenenti un insieme, si può passare al costruttore un riferimento ad un `Set` creato precedentemente, come nel caso di `B1`, oppure creare il `Set` quando si crea la `Lvar`, come nel caso di `B2`.

3.2 Liste: la classe Lst

Una lista è una struttura dati, eventualmente vuota, costituita da una sequenza di elementi che possono essere in numero arbitrario e di tipo diverso.

Per rappresentare le liste useremo come notazione astratta la rappresentazione usata nel Prolog: la lista data dalla sequenza $''a, b, c''$ sarà rappresentata come $[a, b, c]$, così come la lista vuota sarà rappresentata con la notazione [ ]. Nelle liste si tiene conto delle ripetizioni e dell'ordine in cui compaiono gli elementi, quindi, la lista $[1, 2]$ è diversa dalla lista $[2, 1]$ o dalla lista $[1, 2, 2]$.

Nella nostra libreria una lista è una istanza della classe `Lst`. Una `Lst` è una particolare variabile logica e come tale può essere inizializzata o non inizializzata.

Vediamo alcuni esempi di variabili `Lst`:

```
Lst X = new Lst();          // X: lista non inizializzata
Lst v = Lst.vuoto;          // v = []
Lst l = new Lst(4,7);       // l = [4,5,6,7]
```

La lista `X` è una lista non inizializzata, essa quindi non ha nessun valore, ma potrà essere istanziata in seguito, in conseguenza dei vincoli posti su di essa. La lista `v` e `l` sono inizializzate: `v` è vuota, `l` è la lista degli interi contenuti nell'intervallo $4 \div 7$.

E' anche possibile creare una lista passando al costruttore l'array dei suoi elementi nell'ordine in cui devono comparire nella lista:

```
int[] i = {2,4,5,8,3};      // array di interi
Lst w = new Lst(i);         // w = [2,4,5,8,3]

String[] s = {"lista","di","stringhe"}; // array di String
Lst y = new Lst(s);         // y = ["lista","di","stringhe"]
```

In questo modo si possono creare liste inizializzate i cui elementi sono di tipo primitivo, oggetti delle classi wrapper o oggetti `String`.

Le liste inizializzate viste fino a questo momento sono liste i cui elementi sono variabili o oggetti standard di Java. Tra gli elementi delle liste possono però comparire anche delle variabili logiche: ad esempio è possibile costruire delle liste inizializzate del tipo $L_1 = [1, 2, X, 3]$ oppure $L_2 = [X, Y, Z]$ in cui X, Y e Z sono `Lvar` (potrebbero essere anche istanze di `Lst` o `Set`) e come tali, possono essere inizializzate o no. Consideriamo, ad esempio, il caso in cui X sia una `Lvar` non inizializzata, Y sia inizializzata con la lista di caratteri $['a','b']$ e Z sia inizializzata con 4, in questo caso si ha $L_2 = [X, ['a','b'], 4]$ in cui il primo elemento è una variabile logica di cui non si conosce ancora il valore. Le liste L_1 e L_2 sono quindi liste *parzialmente specificate*, in quanto contengono delle variabili non ancora definiti, inoltre L_1 e L_2 sono liste *limitate*, in quanto la loro cardinalità è nota.

La classe `Lst` permette anche di costruire liste parzialmente specificate *non limitate*, cioè con un certo numero di elementi noti, che possono eventualmente essere `Lvar` anche non inizializzate, e un "resto" dato da una lista non inizializzata. Per indicare queste liste useremo, come in Prolog, "|" per dividere la parte nota della lista dalla parte non specificata.

Consideriamo ad esempio la lista $A = [a_1, ..., a_n|R]$ dove con R indichiamo una lista non inizializzata: A è una lista parzialmente specificata non limitata, che non va confusa con la lista $B = [a_1, ..., a_n, R]$ che è anch'essa parzialmente specificata, ma limitata. A è costituita da una parte nota data da n elementi e da un resto R non specificato; B è una lista di $n + 1$ elementi tra i quali figura una lista R non inizializzata. Supponiamo che successivamente la lista R diventi inizializzata, ad esempio, consideriamo $R = [r_1, ..., r_p]$, allora le liste A e B diventano: $A = [a_1, ..., a_n, r_1, ..., r_p]$

e $B = [a_1, ..., a_n, [r_1, ..., r_p]]$ quindi A è una lista di $n+p$ elementi, mentre B continua ad avere $n + 1$ elementi.

Nei prossimi paragrafi vedremo in che modo si possono costruire lista di questo tipo.

3.3 Inserimento ed estrazione di elementi da una lista

La classe `Lst` fornisce diversi metodi `public` per facilitare la creazione e l'utilizzo delle liste. Particolarmente importanti sono quelli che permettono di aggiungere uno o più elementi ad una lista data. Nessuno di questi metodi modifica la lista sulla quale sono invocati, tutti costruiscono e restituiscono una nuova lista ottenuta da quella di partenza aggiungendo in testa o in coda i nuovi elementi passati come parametro.

Altri metodi utili sono quelli che estraggono un elemento dalla testa o dalla coda della lista. Anche in questo caso, la lista su cui sono invocati non viene modificata, ma ne viene creata e restituita una nuova nella quale non compare il primo o l'ultimo elemento della lista di partenza.

Sono previsti i seguenti metodi:

```
public Lst ins1(type elem)
public Lst ins1All(type[] array)
public Lst insn(type elem)
public Lst insnAll(type[] array)
public Lst ext1(Lvar l)
public Lst extn(Lvar l)
```

Per tutti questi metodi *type* può essere: un tipo primitivo, un oggetto wrapper, una stringa, oppure `Lvar`, `Lst` o `Set`.

3.3.1 I metodi ins1 ed ins1All

Il metodo `ins1`, inserisce in testa alla lista un solo elemento, mentre il metodo `ins1All` inserisce in testa tutti gli elementi dell'array passato come parametro, nell'ordine in cui compaiono nell'array.

Consideriamo i seguenti esempi:

```
Lst v = Lst.vuoto;              // v = []
Lst x = new Lst(v.ins1(2));     // x = [2]
Lst y = new Lst(x.ins1(1));     // y = [1,2]
```

La lista `x` è ottenuta aggiungendo alla lista vuota `v` l'intero `2`, quindi `x` è una lista di un solo elemento. La lista `y` è ottenuta aggiungendo alla lista `x` l'intero `1`, quindi `y` è una lista di due elementi.

Il parametro passato al metodo `ins1` può essere anche una variabile logica:

```
Lvar L = new Lvar();            // L: Lvar non inizializzata
Lvar l = new Lvar(4);           // l: Lvar inizializzata a intero

Lst z = new Lst(Lst.vuoto.ins1(L));   // z = [L]
Lst w = new Lst(z.ins1(l));           // w = [4,L]
```

Le liste `z` e `w` sono liste parzialmente specificate limitate. La lista `z` ha come unico elemento una `Lvar` non inizializzata; `t` è una lista di due elementi: un intero e una variabile logica non inizializzata.

I metodi di inserzione ed estrazione possono essere concatenati; infatti essi restituiscono un oggetto `Lst`, e quindi possono essere riapplicati a questo stesso oggetto, l'operazione di concatenazione è associativa a sinistra. Consideriamo ad esempio le liste `y` e `w` degli esempi precedenti, esse possono essere ottenute in modo equivalente come segue:

```
Lst y = new Lst(v.ins1(2).ins1(1));            // y = [1,2]
Lst w = new Lst(Lst.vuoto.ins1(L).ins1(l));    // w = [4,L]
```

Ovviamente la notazione con le `ins1` concatenate è più snella e più leggibile, quindi è sempre conveniente preferirla.

Le variabili logiche passate come parametro al metodo `ins1` possono anche essere delle liste o degli insiemi. Consideriamo ad esempio la seguente lista dove con `w`, `l` e `L` indichiamo le variabili introdotte negli esempi precedenti:

```
Lst t = new Lst(Lst.vuoto.ins1(l).ins1(L).ins1(w).ins1
                // t = [1,[4,L],L,4]
```

la lista `t` è una lista di 4 elementi, di cui il secondo è una lista.

Negli esempi visti sino ad ora abbiamo sempre applicato il metodo `ins1` a liste limitate; in questo modo abbiamo ottenuto delle nuove liste anch'esse limitate.

Il metodo `ins1` può essere applicato anche a liste non inizializzate o non limitate:

```
Lst X = new Lst();           // X: lista non inizializzata
Lvar L = new Lvar();         // L: Lvar non inizializzata
Lvar l = new Lvar(4);        // l: Lvar inizializzata con intero

Lst y = new Lst(X.ins1(L));  // y = [L|X]
Lst z = new Lst(y.ins1(l));  // z = [4,L|X]
```

In questo caso `y` e `z` sono liste parzialmente specificate non limitate. La lista `y` è ottenuta aggiungendo la `Lvar` non inizializzata `L`, alla lista non inizializzata `X`. La lista `z` è ottenuta applicando la `ins1` alla lista non limitata `y`. La lista z poteva essere ottenuta in modo equivalente concatenando le `ins1` come segue:

```
Lst z = new Lst(X.ins1(L).ins1(l));       // z = [4,L|X]
```

La classe `Lst` prevede anche la possibilità di inserire più di un elemento per volta. Il metodo `ins1All` permette di inserire in testa alla lista tutti gli elementi di un array, nell'ordine in cui compaiono nell'array. Il tipo dell'array passato come parametro può essere uno dei tipi previsti per il metodo `ins1`.

Vediamo alcuni esempi di liste costruite con l'ausilio del metodo `ins1All`:

```
int[] i = {1,2};                          // i: array di interi
int[] j = {3,4};                          // j: array di interi
Lst x = new Lst(Lst.vuoto.ins1All(j));    // x = [3,4]
Lst y = new Lst(x.ins1All(i));            // y = [1,2,3,4]
```

La lista `x` è ottenuta aggiungendo alla lista vuota gli elementi dell'array `j`, la lista `y` è ottenuta dalla lista `x` con l'aggiunta, in testa, degli elementi dell'array `i`.

Avremmo potuto costruire la lista `y`, in modo del tutto equivalente, concatenando le `ins1All`:

```
Lst y = new Lst(Lst.vuoto.ins1All(j).ins1All(i));
                    // y = [1,2,3,4]
```

E' anche possibile concatenare tra loro i metodi `ins1` e `ins1All` come nell'esempio che segue:

```
Lst z = new Lst(Lst.vuoto.ins1(3).ins1All(i).ins1(0));
                    // z = [0,1,2,3]
```

Alla lista vuota è stato aggiunto l'intero `3`, a questa nuova lista è stato aggiunto l'array `i` contenente gli interi `1` e `2`, infine alla lista così ottenuta è stato riapplicato il metodo `ins1` ed è quindi stato inserito in testa l'intero `0`.

L'array passato come parametro al metodo `ins1All` può essere anche un array di variabili logiche.
Consideriamo il seguente esempio:

```
Lvar L = new Lvar();          // L: Lvar non inizializzata
Lvar l = new Lvar(2);         // l: Lvar inizializzata con intero
Lvar[] ar = {L,l,m};          // ar: array di Lvar

Lst w = new Lst(Lst.vuoto.ins1All(ar));  // w = [L,2]
```

Il metodo `ins1All` può essere applicato anche a liste non inizializzate o non limitate; le liste così costruite risultano anch'esse non limitate.

Vediamo un esempio:

```
Lst X = new Lst();                // X: lista non inizializzata
int[] i = {1,2};                  // i: array di interi
int[] j = {3,4};                  // j: array di interi
Lst y = new Lst(X.ins1All(j));    // y = [3,4|X]
Lst z = new Lst(y.ins1All(i));    // z = [1,2,3,4|X]
```

Le liste `y` e sono entrambe non limitate, in quanto sono state ottenute applicando il metodo `ins1All` a liste non limitate.

I metodi `ins1` e `ins1All`, così come i metodi che seguono, possono essere applicati, oltre che alle liste, anche a variabili `Lvar` inizializzate con lista.

3.3.2 I metodi insn ed insnAll

A volte può essere utile creare una lista ottenuta da un'altra lista con l'aggiunta di uno o più elementi in coda, anzichè in testa. Per questo scopo si possono utilizzare i metodi `insn` e `insnAll` che aggiungono rispettivamente un solo elemento e tutti gli elementi di un array in coda alla lista. Il tipo dell'elemento o degli elementi aggiunti alla lista può essere uno di quelli visti per i metodi `ins1` e `ins1All`, in particolare si possono quindi aggiungere degli oggetti di tipo `Lvar`, `Lst` o `Set`. Anche i metodi `insn` e `insnAll` restituiscono una `Lst`, e quindi possono essere concatenati tra loro.

Mentre i metodi `ins1` e `ins1All` possono essere invocati su tutte le liste, siano esse inizializzate o no e limitate o no, i metodi `insn` ed `insnAll` sono invece pensati per liste inizializzate e limitate. Questo perchè non è possibile inserire uno o più elementi in coda ad una lista se la sua coda non è istanziata. In realtà se il metodo `insn` viene invocato su una lista non limitata, non si incorre in un errore, ma il metodo restituisce una lista data dalla parte nota della lista di partenza con l'aggiunta del parametro passato, mentre il resto viene trascurato. Quindi i metodi `insn` ed `insnAll` possono essere usati anche in questi casi, ma bisogna tenere presente questo

aspetto, perchè si potrebbero ottenere liste diverse da quelle che si aveva intenzione di creare.

Il risultato dei metodi `insn` ed `insnAll` è sempre una lista limitata anche se vengono invocati su liste non limitate.

Consideriamo i seguenti esempi:

```
Lst X = new Lst();                    // X: Lst non inizializzata
Lst y = new Lst(Lst.vuoto.insn(1));   // y = [1]
Lst z = new Lst(X.insn(1));           // z = [1]
```

Le liste `y` e `z` sono uguali, ma sono ottenute in modo diverso, la lista `y` è ottenuta inserendo l'intero 1 in coda alla lista vuota, la lista `z` è invece ottenuta applicando il metodo `insn` ad una lista non inizializzata; la lista che si ottiene è limitata in quanto la parte non specificata viene trascurata, è quindi consigliabile utilizzare questi metodi solo per liste limitate.

Il metodo `insnAll` è del tutto simile al metodo `insn`, ma permette di inserire in coda alla lista tutti gli elementi dell'array passato, nell'ordine in cui compaiono nell'array.

Vediamo alcuni esempi di liste costruite con l'ausilio del metodo `insnAll`:

```
int[] i = {1,2};                          // i: array di interi
int[] j = {3,4};                          // j: array di interi
Lst x = new Lst(Lst.vuoto.insnAll(i));    // x = [1,2]
Lst y = new Lst(x.insnAll(j));            // y = [1,2,3,4]
```

La lista `x` è ottenuta aggiungendo in coda alla lista vuota gli elementi dell'array `i`, la lista `y` è ottenuta dalla lista `x` con l'aggiunta, sempre in coda, degli elementi dell'array `j`.

Anche le `insnAll` possono essere concatenate. La lista `y`, ad esempio, poteva essere ottenuta come segue:

```
Lst y = new Lst(Lst.vuoto.insnAll(i).insnAll(j));// y =[1,2,3,4]
```

Consideriamo ora il seguente esempio, in cui `i` e `j` sono gli stessi array introdotti nell'esempio precedente:

```
Lst X = new Lst();                    // X: Lst non inizializzata
Lst w = new Lst(X.insnAll(i).insnAll(j));     // w = [1,2,3,4]
```

La lista `w` è limitata anche se ottenuta applicando il metodo `insnAll` ad una lista non inizializzata. Il metodo `insnAll`, come già detto, si comporta come il metodo `insn`: se applicato ad una lista non limitata trascura la parte non nota della lista e quindi restituisce sempre una lista limitata.

L'array passato come parametro al metodo `insnAll` può essere anche un array di variabili logiche. Inoltre i metodi `insn` e `insnAll`, possono essere concatenati tra loro e con i metodi `ins1` e `ins1All`. Ad esempio:

```
int[] i = {1,2};                                  // i: array di interi
int[] j = {4,5};                                  // j: array di interi
Lst z = new Lst(Lst.vuoto.insn(3).ins1All(i).ins1(0).insnAll(j));
                                                  // z = [0,1,2,3,4,5]
```

3.3.3 I metodi ext1 ed extn

La classe `Lst` fornisce anche la possibilità di creare una nuova lista estraendo il primo o l'ultimo elemento da una lista data. I metodi `ext1` e `extn` hanno come tipo di ritorno `Lst` e non modificano la lista sulla quale sono invocati; essi non possono essere applicati a lista vuote o non inizializzate: se si tenta di farlo viene sollevata un'eccezione `EmptyLstException` e la computazione termina.

Il parametro passato a questi metodi deve essere una `Lvar` non inizializzata alla quale sarà assegnato l'elemento estratto dalla lista. In questo modo è possibile conoscere ed utilizzare tale elemento.

Nel caso in cui la `Lvar` passata come parametro alla `ext1` o alla `extn` sia inizializzata, viene generata un'eccezione `InizLvarException` e la computazione fallisce.

Il metodo `ext1`, estrae il primo elemento in testa alla lista, mentre il metodo `extn` estrae l'ultimo elemento della lista .

Vediamo alcuni esempi dell'utilizzo del metodo `ext1`:

```
Lvar L1 = new Lvar();              // L1: Lvar non inizializzata
Lvar L2 = new Lvar();              // L2: Lvar non inizializzata
int[] i = {1,2,3};                 // i: array di interi
Lst l = new Lst(i);                // l = [1,2,3]
Lst y = new Lst(l.ext1(L1));       // y = [2,3]
Lst z = new Lst(y.ext1(L2));       // z = [3]
```

La lista `y` è ottenuta dalla lista `l` estraendo il primo elemento. La lista `z` è ottenuta riapplicando il metodo alla lista `y`. A `L1` viene assegnato il valore `1` e a `L2` il valore 2.

Anche i metodi `ext1` e `extn` possono essere concatenati. Ad esempio, la lista `z`, dell'esempio appena considerato, poteva essere ottenuta, in modo equivalente, come segue:

```
Lst z = new Lst(l.ext1().ext1());   // z = [3]
```

Il metodo `extn`, invece, estrae l'ultimo elemento della lista. Come i metodi `insn` e `insnAll`, è indicato per liste limitate in quanto non è possibile estrarre un elemento dalla coda di una lista se la coda non è nota. Se viene applicato ad una lista non limitata, non viene sollevata nessuna eccezione e non si incorre in nessun errore, ma la lista che viene restituita è ottenuta considerando solo la parte nota della lista e trascurando invece il resto, quindi il metodo restituisce sempre una lista limitata anche se applicato a liste che non lo sono. E' comunque consigliabile usare il metodo `extn` solo con liste limitate.

Vediamo alcuni esempi di liste ottenute con l'uso di `extn`:

```
int[] i = {1,2,3};                 // i: array di interi
Lst l = new Lst(i);                // l = [1,2,3]
```

```
Lst w = new Lst(l.extn());            // w = [1,2]
Lst t = new Lst(y.extn());            // t = [1]
```

La lista `w` è ottenuta dalla lista `l` estraendo l'ultimo elemento. La lista `t` è ottenuta riapplicando il metodo alla lista `w`. La lista `t` poteva essere ottenuta in modo equivalente concatenando le `extn`:

```
Lst w = new Lst(l.extn().extn());     // w = [1]
```

I metodi `ext1` e `extn`, come i metodi di inserimento, possono essere concatenati tra loro. Inoltre è possibile ottenere nuove liste concatenando i metodi di inserimento con quelli di estrazione:

```
int[] i = {1,2};                      // i: array di interi
int[] j = {4,5};                      // j: array di interi
Lst z = new Lst(Lst.vuoto.ins1(0).insnAll(i).extn().insnAll(j));
                                      // z = [0,1,4,5]
```

Inizialmente alla lista vuota è stato aggiunto (in testa) l'intero 0, a questa nuova lista è stato aggiunto, in coda, l'array `i` contenente gli interi `1` e `2` ottenendo quindi la lista *[0,1,2]*, a questa lista è stato poi estratto l'ultimo elemento, infine alla lista *[0,1]* così ottenuta, è stato applicato il metodo `insnAll` per inserire in coda gli elementi dell'array `j`.

3.4 Insiemi: la classe Set

Un insieme è una struttura dati simile ad una lista ma nella quale l'ordine degli elementi e le ripetizioni non sono significativi. Un insieme può essere vuoto o costituito da un numero arbitrari di elementi non necessariamente dello stesso tipo.

Per rappresentare gli insiemi useremo una notazione del tutto simile a quella usata per le liste, ma usando le parentesi graffe invece delle quadre; ad esempio,

l'insieme dato dagli elementi $''a, b, c''$ sarà rappresentato come $\{a, b, c\}$, e l'insieme vuoto sarà rappresentato con la notazione $\{\,\}$.

Nella nostra libreria un insieme è una istanza della classe `Set`. Anche la classe `Set`, come la classe `Lst`, permette di definire insiemi non inizializzati o inizializzati, quindi anche i `Set` sono particolari variabili logiche, i cui possibili valori sono gli insiemi.

I costruttori della classe `Set` sono del tutto simili a quelli visti per la classe `Lst`. Vediamo alcuni esempi di variabili Set:

```
Set X = new Set();              // X: insieme non inizializzato
Set v = Set.vuoto;              // v = {}
Set i = new Set(4,7);           // i = {4,5,6,7}
```

L'insieme `X` è non inizializzato, esso quindi non ha ancora nessun valore, ma potrà essere istanziato in seguito, in conseguenza dei vincoli posti su di esso. Gli insiemi `v` e `i` sono inizializzati: `v` è l'insieme vuoto, `i` è l'insieme degli interi contenuti nell'intervallo $4 \div 7$.

E' anche possibile creare un insieme passando al costruttore l'array dei suoi elementi:

```
int[] i = {2,4,5,8,3};      // i: array di interi
Set w = new Set(i);         // w = {2,4,5,8,3}

String[] s = {"insieme","di","stringhe"}; // s: array di String
Set y = new Set(s);         // y = {"insieme","di","stringhe"}
```

In questo modo, come per le liste, si possono creare insiemi inizializzati i cui elementi sono di tipo primitivo, oggetti delle classi wrapper o oggetti `String`.

Tra gli elementi di un insiemi possono comparire delle variabili logiche generiche o anche delle liste e degli insiemi, ad esempio è possibile costruire gli insiemi $S_1 = \{1, 2, X, 3\}$ e $S_2 = \{X, Y, Z\}$ con X, Y e Z istanze `Lvar`, `Lst` o `Set`. Se tra gli elementi di un insieme ci sono delle variabili logiche non inizializzate, l'insieme

è *parzialmente specificato*. In questo caso la cardinalità dell'insieme può variare in base al valore assunto dalle variabili che vi compaiono: ad esempio, se la variabile X viene inizializzata col valore 1, l'insieme S_1 ha cardinalità 3; infatti diventa $S_1 = \{1, 2, 1, 3\} = \{1, 2, 3\}$, mentre se X assume un valore diverso dagli altri elementi dell'insieme, la sua cardinalità è 4. Analogamente, la cardinalità di S_2 può variare tra 1 e 3.

Gli insiemi visti fino ad ora sono insiemi *limitati*, in quanto la loro cardinalità è superiormente limitata, anche se varia in base al valore delle variabili logiche che vi compaiono.

La classe `Set` permette anche di costruire insiemi *non limitati*, cioè con un certo numero di elementi noti, che possono eventualmente essere `Lvar` e quindi anche delle liste o degli insiemi, e un "resto" dato da un insieme non inizializzato. Nell' indicare questi insiemi useremo la stessa notazione adottata per le liste. Ad esempio l'insieme $A = \{a_1, ..., a_n | R\}$ con R insieme non inizializzato è un `Set` non limitato, diverso dal `Set` $B = \{a_1, ..., a_n, R\}$ che invece è limitato. Supponiamo che successivamente l'insieme R diventi inizializzato, ad esempio $R = \{r_1, ..., r_p\}$, allora gli insiemi A e B diventano: $A = \{a_1, ..., a_n, r_1, ..., r_p\}$ e $B = \{a_1, ..., a_n, \{r_1, ..., rp\}\}$, A è un insieme con al più $n + p$ elementi, B invece ha al più $n + 1$ elementi.

La realizzazione di questi insiemi, come vedremo, è del tutto simile alla realizzazione delle liste dello stesso tipo.

3.5 Inserimento di elementi in un insieme

La classe `Set` fornisce alcuni metodi `public` per facilitare la creazione e l'utilizzo degli insiemi. Come abbiamo già detto, l'ordine degli elementi nei `Set` non è importante, quindi non è previsto un metodo di inserimento in testa e uno in coda, perchè produrrebbero lo stesso insieme. Sono previsti solo due metodi: uno per inserire nell'insieme un solo elemento e l'altro per inserire tutti gli elementi di un array. Tali metodi sono:

```
public Set ins(type elem)
public Set insAll(type [] arr)
```

Questi metodi non modificano l'insieme sul quale sono invocati, ma costruiscono e restituiscono un nuovo insieme; *type* può essere: `char`, `boolean`, `int`, `byte`, `short`, `long`, `float`, `double`, `Character`, `Boolean`, `Integer`, `Byte`,...,`String`, `Lvar`, `Lst`, `Set`.

Il metodo `ins`, inserisce nell'insieme un solo elemento, mentre il metodo `insAll` inserisce tutti gli elementi dell'array passato come parametro.

Consideriamo i seguenti esempi:

```
Set v = Set.vuoto;              // v = {}
Set x = new Set(v.ins(2));      // x = {2}
Set y = new Set(x.ins(1));      // y = {1,2}
Set z = new Set(y.ins(1));      // z = {1,1,2} = {1,2}
```

L'insieme `x` è ottenuta aggiungendo all'insieme vuoto `v` l'intero `2`, l'insieme `y` è ottenuto aggiungendo all'insieme `x` l'intero `1`, l'insieme `z` è identico all'insieme `y` perchè è ottenuto aggiungendo all'insieme `y` l'intero `1` che apparteneva già all'insieme.

Il parametro passato al metodo `ins` può essere anche una variabile logica:

```
Lvar L = new Lvar();    // L: Lvar non inizializzata
Lvar l = new Lvar(4);   // l: Lvar inizializzata con intero

Set t = new Set(Set.vuoto.ins(L));   // t = {L}
Set w = new Set(t.ins(l));           // w = {4,L}
```

L'insieme `t` ha come unico elemento una `Lvar` non inizializzata; `w` è un insieme ottenuto aggiungendo a `t` una `Lvar` inizializzata con intero.

I metodi `ins` e `insAll` possono essere concatenati; infatti essi restituiscono un oggetto `Lst`, e quindi possono essere riapplicati a questo stesso oggetto. Consideriamo ad esempio gli insiemi `y` e `w` degli esempi precedenti, essi possono essere ottenute in modo equivalente concatenando le `ins`:

```
Set y = new Set(v.ins(2).ins(1));              // y = {1,2}
Set w = new Set(Set.vuoto.ins(l).ins(L));      // w = {L,4}
```

Ovviamente la notazione con le `ins` concatenate è più snella e più leggibile, quindi è sempre conveniente preferirla.

Le variabili logiche passate come parametro al metodo `ins` possono anche essere delle liste o degli insiemi. Consideriamo ad esempio il seguente insieme dove con `w`, `l` e `L` indichiamo le variabili introdotte negli esempi precedenti e con `m` indichiamo la lista `[5,6]`:

```
Set t = new Set(Set.vuoto.ins(l).ins(L).ins(w).ins(m));
                // u = {[5,6],{L,4},L,4}
```

l'insieme `u` ha un elemento di tipo `Lst`, un elemento di tipo `Set`, una `Lvar` non inizializzata e una inizializzata con l'intero `4`.

Negli esempi visti sino ad ora abbiamo sempre applicato il metodo `ins` a insiemi limitati; in questo modo abbiamo ottenuto dei nuovi insiemi anch'essi limitati.

Il metodo `ins` può essere applicato anche a insiemi non limitati:

```
Set X = new Set();          // X: insieme non inizializzato
Lvar L = new Lvar();        // L: Lvar non inizializzata
Lvar l = new Lvar(4);       // l: Lvar inizializzata con intero

Set y = new Set(X.ins(L));  // y = {L|X}
Set z = new Set(y.ins(l));  // z = {4,L|X}
```

In questo caso `y` è ottenuto aggiungendo la `Lvar` non inizializzata `L`, alla insieme non inizializzato `X`, quindi `y` è un insieme parzialmente specificato e non limitato. L'insieme `z` è ottenuto applicando la `ins` all'insieme `y`; anch'esso è parzialmente specificato e non limitato, in quanto lo è l'insieme da cui è stato ottenuto.

L' insieme `z` poteva essere ottenuto in modo equivalente concatenando le `ins` come segue:

```
Set z = new Set(X.ins(L).ins(l));          // z = {4,L|X}
```

La classe `Set` prevede anche la possibilità di inserire più di un elemento per volta. Il metodo `insAll` permette di inserire nell'insieme tutti gli elementi dell'array passato come parametro.

Vediamo alcuni esempi di insiemi costruiti con l'ausilio del metodo `insAll`:

```
int[] i = {1,2};              // i: array di interi
Lvar L = new Lvar();          // L: Lvar non inizializzata
Lvar l = new Lvar(2);         // l: Lvar inizializzata con intero
Lvar[] ar = {L,l,m};          // ar: array di Lvar
Set x = new Set(set.vuoto.insAll(j)); // x = {2,3}
Set y = new set(x.insAll(ar));        // y = {L,2,2,3} = {L,2,3}
```

L'insieme `x` è ottenuto aggiungendo all'insieme vuoto gli elementi dell'array `j`, l'insieme `y` è ottenuto da `x` con l'aggiunta degli elementi dell'array `ar`.

Avremmo potuto costruire l'insieme `y`, in modo del tutto equivalente, concatenando le `insAll`:

```
Set y = new Set(Set.vuoto.insAll(j).insAll(i));  // y = {1,2,3}
```

Il metodo `insAll` può essere applicato anche a insiemi non limitati; gli insiemi così costruiti risultano anch'essi non limitati.

Vediamo alcuni esempi:

```
Set X = new Set();              // X: lista non inizializzata
int[] i = {1,2};                // i: array di interi
int[] j = {2,3};                // j: array di interi
Set y = new Lst(X.insAll(j));   // y = {2,3|X}
Set z = new Lst(y.insAll(i));   // z = {1,2,3|X}
```

In questo caso `y` è ottenuto aggiungendo un array di interi all'insieme non inizializzato `X`, quindi `y` è un insieme non limitato.

L'insieme `z` è ottenuto applicando la `insAll` a `y`, anch'esso è non limitato, in quanto lo è l'insieme da cui è stato ottenuto. L'insieme `z` poteva essere ottenuto in modo equivalente concatenando le `insAll` come mostrato di seguito:

```
Set z = new Set(X.insAll(j).insAll(i));   // z = {1,2,3|X}
```

I metodi `ins` e `insAll` possono essere applicati, oltre che agli oggetti della classe `Set`, anche a variabili `Lvar` inizializzate con insieme. Se la `Lvar` a cui vengono applicati non è inizializzata con insieme, viene generata un'eccezione `NotSetException` e la computazione termina.

La classe `Set` non prevede metodi di estrazione come quelli visti per le liste, infatti l'estrazione del primo o dell'ultimo elemento non sarebbe sensata visto che gli insiemi non sono strutture dati ordinate. Inoltre, l'estrazione di uno o più elementi da un insieme può essere realizzata dai vincoli `less` e `differ`: il vincolo `less` permette di estrarre un elemento da un insieme, se l'elemento non appartiene all'insieme si ha un fallimento. Il vincolo `differ` esegue la differenza tra due insiemi. Il vincolo `less` non può essere applicato all'insieme vuoto, ma a parte questa, non ci sono altre restrizioni all'uso di questi due vincoli, cioè possono essere applicati ad un qualunque insieme: completamente o parzialmente specificato, limitato o non limitato.

L'estrazione, nel caso degli insiemi, è realizzata attraverso i vincoli, in quanto, in generale, comporta una scelta non deterministica. Si consideri, ad esempio, di voler estrarre dall'insieme $\{1, 2, 3\}$ la `Lvar` non inizializzata X; in questo caso la soluzione non è unica, il valore di X può essere 1, 2 o 3. Analogamente, l'insieme ottenuto come differenza tra $\{1, 2, 3\}$ e $\{X\}$ è l'insieme $\{1, 2, 3\}$ nel caso in cui X sia diverso 1, 2 e 3, negli altri casi è l'insieme ottenuto da $\{1, 2, 3\}$ estraendo X. Inoltre anche l'estrazione di un elemento noto può comportare una scelta non deterministica. Consideriamo ad esempio di voler estrarre l'intero 1 dall'insieme $\{1, 2, X\}$, in questo caso l'insieme che si ottiene è l'insieme $\{2\}$ nel caso in cui ad X sia assegnato il valore 1 o 2, mentre è l'insieme $\{2, X\}$ negli altri casi. La stessa cosa vale per l'insieme ottenuto come differenza tra $\{1, 2, X\}$ e $\{1\}$.

L'estrazione di uno (o più elementi) da un insieme è deterministica solo nel caso in cui l'elemento sia noto, l'insieme sia limitato e tutti i suoi elementi siano noti, negli altri casi comporta una scelta non deterministica.

3.6 Metodi di utilità della classe Lst

La classe `Lst` fornisce alcuni metodi di utilità. Tali metodi sono usati per verificare se un elemento appartiene oppure no ad una lista, per concatenare due liste, per leggerne gli elementi, per determinarne la lunghezza, per la stampa o per creare una lista di n `Lvar` non inizializzate.

Di seguito riportiamo i metodi di utilità della classe `Lst`:

```
public boolean inl(type elem)
public boolean ninl(type elem)
public Lst concat(Lst l)
public Object get(int i)
public Object size()
public static void output(Lst l)
public static Lst mkLst(int n)
```

3.6.1 I metodi inl e ninl

I metodi `inl` e `ninl` restituiscono un booleano: il metodo `inl` restituisce true se `elem` è un elemento della lista, e false altrimenti; viceversa `ninl` restituisce true se `elem` non è un elemento della lista, e false altrimenti. Anche per questi metodi, l'elemento passato come parametro può essere di tipo primitivo, oppure può essere un'istanza di una classe wrapper, o della classe `String`, oppure un oggetto `Lvar`, `Lst` o `Set`.

I metodi `inl` e `ninl` possono essere eseguiti solo nel caso in cui la lista su cui vengono invocati e il parametro passato siano completamente specificati; se al momento dell'esecuzione queste condizioni non sono rispettate, viene sollevata un eccezione

`NotInizVarException` e la computazione termina. Consideriamo ad esempio di avere una lista `L` e una `Lvar x`, con `L` completamente specificata e `x` inizializzata; il metodo `L.inl(x)` restituisce true, se `L` contiene un elemento il cui valore coincide con il valore di `x`, false altrimenti. Analogamente il metodo `L.inl(y)`, con `y` istanza della classe `Set` (o `Lst`), restituisce true se `y` è un elemento della lista oppure se in `L` c'è un oggetto `Set` (o `Lst`) identico a `y`.

I metodi `inl` e `ninl` possono essere invocati anche su variabili `Lvar` inizializzate e il cui valore sia un oggetto `Lst`. Se questi metodi sono invocati su una `Lvar` che non soddisfa queste condizioni, viene generata un'eccezione `NotLstException`, e la computazione termina.

3.6.2 Il metodo concat

Date le liste $l1 = [a1, ..., an]$ e $l2 = [b1, ..., bm]$, l'istruzione `l1.concat(l2)` restituisce la lista $[a1, ..., an, b1, ..., bm]$ e lascia invariata la lista su cui è invocata e la lista passata come parametro. Questo metodo è pensato per liste completamente specificate, infatti se applicato a liste non completamente istanziate, il metodo trascura i resti e quindi non restituisce la lista ottenuta dalla concatenazione delle due liste, ma una lista completamente istanziata in cui compaiono gli elementi contenuti nella parte specificata di ciascuna delle due liste; quindi la lista restituita dall'`concat` è sempre una lista completamente specificata.

Riportiamo di seguito alcuni esempi di utilizzo del metodo `concat`:

```
int[] i = {1,2,3};              // i: array di interi
boolean[] b = {true, false};    // b: array di boolean
Lst l = new Lst(i);             // l = [1,2,3]
Lst m = new Lst(b);             // m = [true,false]
Lst X = new Lst();              // X: Lst non inizializzata

Lst y = new Lst(l.concat(m));            // y = [1,2,3,true,false]
Lst z = new Lst(X.ins1All(i).concat(m)); // z = [1,2,3,true,false]
```

Le liste `y` e `z` sono identiche, anche se ottenute in modo diverso. La lista `y` è ottenuta concatenando due liste completamente istanziate, cioè la lista `[1,2,3]` e la lista `[true,false]`. La lista `z` è ottenuta applicando il metodo `concat` alla lista parzialmente specificata `[1,2,3|X]` che a sua volta è stata ottenuta applicando il metodo `ins1All` alla lista non inizializzata `X`; la lista restituita dal metodo `concat` è completamente specificata in quanto come si è detto, se applicata a liste parzialmente specificate, essa trascura i resti. Se in seguito, la lista `X` diventasse istanziata in conseguenza dei vincoli posti su di essa, la lista `z` non verrebbe modificata; è quindi consigliabile utilizzare questo metodo solo per liste completamente definite.

3.6.3 I metodi get e size

Il metodo `get` serve per l'accesso agli elementi di una lista in base alla loro posizione. Data una lista `l`, l'istruzione `l.get(i)` restituisce l'elemento che si trova in i-esima posizione nella lista `l`. Se l'indice passato al metodo get è minore di 0 o è maggiore o uguale alla dimensione della lista, viene sollevata un'eccezione `ArrayIndexOutOfBoundsException`.

Il metodo `size()` restituisce la dimensione della lista alla quale è applicato, cioè il numero di elementi nella parte istanziata. Ad esempio, se con `l` indichiamo la lista `[1,2,3|X]`, `l.size()` restituisce l'intero 3.

Il metodo `size`, se applicato ad una lista vuota o non inizializzata, restituisce il valore 0.

3.6.4 Il metodo output

Il metodo statico `output` è il metodo che deve essere usato per la stampa di una lista. Per la stampa di una lista `l` si userà il comando: `Lst.output(l)`.

Supponiamo che `l` sia la lista `[1,2,3]` allora il comando:

```
System.out.print("Lst l = ");Lst.output(l);
```

produrrà l'output:

```
Lst l = [1,2,3]
```

Supponiamo invece che la lista `l` sia parzialmente specificata, ad esempio, `l = [1,2,3|X]`, con `X` lista non inizializzata. Allora lo stesso comando produrrebbe:

```
Lst l = [1,2,3|Lst@53c015]
```

dove `Lst@53c015` è l'identificatore usato dalla JVM per memorizzare la lista non inizializzata `X`.

Anche le classe `Lvar` e `Set`, come vedremo, prevedono un metodo statico `output` per la stampa delle loro istanze.

3.6.5 Il metodo mkLst

Il metodo statico `mkLst` restituisce una lista di n `Lvar` non inizializzate, dove n è l'intero passato come parametro al metodo.

3.7 Metodi di utilità della classe Set

La classe `Set` fornisce alcuni metodi di utilità, tali metodi sono usati per concatenare due insiemi, per la stampa e per creare un insieme di n `Lvar` non inizializzate.

Di seguito sono elencati i metodi di utilità della classe `Set`:

```
public Set concat(Set s)
public static void output(Set l)
public static Set mkSet(int n)
```

3.7.1 Il metodo concat

Dati due insiemi $s1 = \{a1, ..., an\}$ e $s2 = \{b1, ..., bm\}$, l'istruzione `s1.concat(s2)` restituisce l'insieme $\{a1, ..., an, b1, ..., bm\}$ e lascia invariati l' insieme su cui è invocata e quello passato come parametro. Questo metodo è pensato per insiemi

completamente specificati, infatti se applicato a insiemi non completamente istanziati, il metodo trascura i resti e quindi non restituisce l'insieme ottenuto dalla concatenazione dei due insiemi dati, ma un insieme completamente istanziato in cui compaiono gli elementi contenuti nella parte specificata di ciascuno dei due insiemi; quindi l'insieme restituito dalla `concat` è sempre un insieme completamente specificato.

Riportiamo di seguito alcuni esempi di utilizzo del metodo `concat`:

```
int[] i = {1,2,3};                      // i: array di interi
boolean[] b = {true, false};            // b: array di boolean
Set s = new Set(i);                     // s = {1,2,3}
Set t = new Set(b);                     // t = {true,false}
Set X = new Set();                      // X: Set non iniz.

Set y = new Set(s.concat(t));           // y = {1,2,3,true,false}
Lst z = new Lst(X.insAll(i).concat(t));// z = {1,2,3,true,false}
```

Gli insiemi `y` e `z` sono identici, anche se ottenuti applicando la `concat` a insiemi potenzialmente diversi. L'insieme `y` è ottenuto concatenando due insiemi completamente istanziati, cioè l'insieme {`1,2,3`} e l'insieme {`true,false`}. L'insieme `z` è ottenuto applicando il metodo `concat` all'insieme parzialmente specificato {`1,2,3|X`} che a sua volta è stata ottenuto applicando il metodo `insAll` all'insieme non inizializzato `X`; l'insieme restituito dal metodo `concat` è completamente specificato in quanto sono stati trascurati i resti. Se in seguito, l'insieme `X` diventasse istanziato in conseguenza dei vincoli posti su di esso, `z` non verrebbe modificato; è quindi consigliabile utilizzare questo metodo solo per insiemi completamente definiti.

3.7.2 Il metodo **output**

Il metodo statico `output` è il metodo che deve essere usato per la stampa di un insieme. Ad esempio, per la stampa di un insieme `s` si userà il comando: `Set.output(s)`.

Supponiamo che `s` sia l'insieme {`1,2,3`} allora il comando:

```
System.out.print("Set s = ");Set.output(s);
```

produrrà l'output:

```
Set s = {1,2,3}
```

Supponiamo invece che l'insieme `s` sia parzialmente specificato, ad esempio, `s = {1,2,3|X}`, con `X` insieme non inizializzato. Allora lo stesso comando produrrebbe l'output:

```
Set s = {1,2,3|Set@53c015}
```

dove `Set@53c015` è l'identificatore usato dalla JVM per memorizzare l'insieme non inizializzato `X`.

3.7.3 Il metodo mkSet

Il metodo statico `mkSet` restituisce un insieme di n `Lvar` non inizializzate, dove n è l'intero passato come parametro al metodo.

Capitolo 4

I vincoli

4.1 Introduzione

In `JavaSet` un vincolo è una relazione binaria o ternaria, definita mediante le espressioni:

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

dove e_1 è un'espressione di tipo `Lvar`, `Lst` o `Set`, mentre e_2 ed e_3, a seconda del vincolo definito su di esse, possono essere espressioni di tipo primitivo, wrapper, `String`, `Lvar`, `Lst` o `Set`. La congiunzione di due o più vincoli è anch'esso un vincolo. L'espressione che segue definisce un vincolo ottenuto come congiunzione dei vincoli v_1, v_2,...,v_n:

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

Ad esempio, siano `X`, `Y` e `Z` tre variabili logiche e `R`, `S`, `T` tre insiemi. Le espressioni che seguono sono vincoli:

```
R.eq(S)
T.union(R,S)
X.eq(Y).and(X.eq(3)).and(Y.neq(Z))
```

Il primo è un vincolo di uguaglianza tra gli insiemi `R` e `S`. Il secondo definisce il vincolo di unione: $T = R \cup S$. Il terzo è un vincolo dato dalla congiunzione di tre vincoli: due di uguaglianza e uno di disuguaglianza.

L'introduzione di un vincolo avviene principalmente tramite il metodo `add` della classe `Solver`, cioè l'espressione $Solver.add(v)$ introduce il vincolo v nello store. Altri metodi (`AllDifferent` e `forall`) saranno esaminati più avanti (sezione 4.12.1 pag. 85). Ad esempio, per introdurre i vincoli appena definiti sono necessarie le istruzioni:

```
Solver.add(X.eq(Y));
Solver.add(T.union(R,S));
Solver.add(X.eq(Y).and(X.eq(3)).and(Y.neq(Z)))
```

In questo modo tutti i vincoli vengono memorizzati in un *"contenitore"* nella classe `Solver` detto *"constraint store"*.

La risoluzione dei vincoli avviene tramite la seguente istruzione:

```
Solver.solve();
```

Il metodo `solve` della classe `Solver` ricerca, in modo non deterministico, una soluzione che soddisfi tutti i vincoli presenti nel *"constraint store"*. Se non viene trovata nessuna soluzione la computazione fallisce, cioè viene generata un'eccezione `NoSolutionFound` che provoca la terminazione del thread corrente.

L'approccio adottato in JavaSet per la risoluzione dei vincoli è lo stesso utilizzato in SINGLETON [16] e sviluppato in CLP($\mathcal{SET}$) [8]. Anche i vincoli definiti in JavaSet sono gli stessi di CLP($\mathcal{SET}$) e di SINGLETON: uguaglianza, appartenenza, inclusione, unione, disgiunzione, intersezione, differenza e le loro negazioni.

Sia l_1 un espressione di tipo `Lvar`, `Lst` o `Set`, l_2 un'espressione di tipo primitivo, wrapper, `String`, `Lvar`, `Lst` o `Set`, m_1 una `Lvar` inizializzata con intero, ed m_2 una `Lvar` inizializzata con intero o un'espressione di tipo `int` o `Integer`. Infine siano s_1, s_2 e s_3 espressioni di tipo `Set`. Quelli che seguono sono i vincoli definiti in JavaSet:

- **Uguaglianza** e **disuguaglianza**: $l_1.eq(l_2)$, $l_1.neq(l_2)$
- **Appartenenza** e **non appartenenza** ad un insieme: $l_1.in(s_1)$, $l_1.nin(s_1)$
- Vincoli di **confronto**: $m_1.le\,(m_2)$, $m_1.lt\,(m_2)$, $m_1.ge\,(m_2)$, $m_1.gt\,(m_2)$
- Vincoli **aritmetici**: $m_1.sum\,(m_2)$, $m_1.sub\,(m_2)$, $m_1.mul\,(m_2)$, $m_1.div(m_2)$
- **Disgiunzione** e **non disgiunzione**: $s_1.disj(s_2)$, $s_1.ndisj(s_2)$
- **Unione** e **non unione**: $s_1.union(s_2, s_3)$, $s_1.nunion(s_2, s_3)$
- **inclusione** e **non inclusione**: $s_1.subset(s_2)$, $s_1.nsubset(s_2)$
- **intersezione** e **non intersezione**: $s_1.inters(s_2, s_3)$, $s_1.ninters(s_2, s_3)$
- **differenza** e **non differenza** tra insiemi: $s_1.differ(s_2, s_3)$, $s_1.ndiffer(s_2, s_3)$
- Vincolo ***less***: $s_1.less(l_1, s_2)$

Ai vincoli appena elencati, dobbiamo poi aggiungere i metodi `allDifferent` e `forall`. Questi metodi non sono vincoli, ma di fatto, durante la loro esecuzione, aggiungono dei nuovi vincoli al constraint store.

4.2 Vincoli di uguaglianza e disuguaglianza

Le espressioni $l_1.eq(l_2)$ e $l_1.neq(l_2)$ sono rispettivamente un vincolo di uguaglianza e un vincolo di disuguaglianza tra l_1 e l_2; l_1 può essere un'espressione di tipo `Lvar`, `Lst` o `Set`, l_2 può essere di tipo primitivo, un oggetto delle classi wrapper, un oggetto `String`, oppure un'espressione di tipo `Lvar`, `Lst` o `Set`.

Vediamo alcuni semplici esempi in cui vengono introdotti e risolti dei vincoli di uguaglianza e disuguaglianza:

Esempio 2

```
class EqNeq1 {
   public static void main(String[]args){
      Lvar X = new Lvar();          // X: Lvar non inizializzata
      Lvar Y = new Lvar();          // Y: Lvar non inizializzata
      Lvar Z = new Lvar();          // Z: Lvar non inizializzata

      Solver.add(X.eq(3));          // vincolo di uguaglianza
      Solver.add(Y.neq(X));         // vincolo di disuguaglianza
      Solver.add(X.eq(Z));          // vincolo di uguaglianza

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

La classe `EqNeq1` crea tre variabili logiche non inizializzate, poi vengono introdotti un vincolo di uguaglianza tra `X` e l'`int 3`, un vincolo di disuguaglianza tra `X` e `Y` e un vincolo di uguaglianza tra `X` e `Z`. Quindi, in termini astratti, sono stati aggiunti i vincoli: $X = 3 \wedge X \neq Y \wedge Z$.

Le espressioni:

```
Solver.add(X.eq(3));
Solver.add(Y.neq(X));
Solver.add(X.eq(Z));
```

sono equivalenti all'istruzione:

```
Solver.add(X.eq(3).and(Y.neq(X)).and(X.eq(Z)));
```

I tre vincoli introdotti sono soddisfacibili. Al termine dell'esecuzione del metodo `solve`, le variabili logiche `X` e `Z` hanno entrambe il valore `3` mentre `Y` resta non inizializzata. □

Consideriamo ora il seguente esempio di vincoli non soddisfacibili:

Esempio 3

```
class EqNeq2 {
   public static void main(String[]args) {
     Lvar X = new Lvar();        // X: Lvar non inizializzata
     Lvar Y = new Lvar(1);       // Y: Lvar inizializzata con 1

     Solver.add(X.neq(1));       // vincolo di disuguaglianza
     Solver.add(Y.eq(X));        // vincolo di uguaglianza

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

La classe `EqNeq2` crea le due variabili logiche `X` e `Y`: `X` è una `Lvar` non inizializzata, mentre `Y` è inizializzata con `1`. In termini astratti i vincoli introdotti sono: $X \neq 1 \wedge Y = X$. Questi due vincoli sono incompatibili, in quanto il valore di `X` dovrebbe essere diverso da `1`, ma `X` dovrebbe anche essere uguale a `Y` che è inizializzata proprio con `1`. In questo caso viene sollevata un'eccezione `NoSolutionFound` e la computazione termina. □

Come abbiamo già detto, le variabili logiche coinvolte nei vincoli di uguaglianza e disuguaglianza possono anche essere istanze delle classi `Lst` e `Set` oppure possono essere `Lvar` inizializzate con lista o con insieme.

Vediamo alcuni esempi:

Esempio 4

```
class EsempiEqNeq {
  public static void main(String[]args){
```

```
        Lvar X = new Lvar();        // X: Lvar non inizializzata
        Lvar Y = new Lvar();        // Y: Lvar non inizializzata
        Lvar Z = new Lvar();        // Z: Lvar non inizializzata

        int[] s_val = {1,2,3};
        Set s = new Set(s_val);     // s = {1,2,3}

        Solver.add(X.eq(3));        // vincolo di uguaglianza
        Solver.add(Y.neq(X));       // vincolo di disuguaglianza
        Solver.add(Z.eq(Y));        // vincolo di uguaglianza
        Solver.add(Z.eq(s));        // vincolo di uguaglianza

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

In questo esempio vengono create tre `Lvar` non inizializzate `X`, `Y` e `Z` ed un `Set` inizializzato `s = {1,2,3}`, in termini astratti i vincoli introdotti sono: $X = 3 \wedge Y \neq X \wedge Z = Y \wedge Z = \{1, 2, 3\}$; al termine del metodo `solve()` le tre `Lvar` sono inizializzate; il valore di `X` è `3`, mentre il valore di `Y` e `Z` è uguale al `Set s`.

Se sostituiamo il vincolo `Y.neq(X)` con `Y.neq(Z)` il problema non ha nessuna soluzione, in quanto i vincoli `Y.neq(Z)` e `Z.eq(Y)` sono incompatibili. In questo caso viene sollevata un'eccezione `NoSolutionFound` e la computazione termina. □

In generale i vincoli di uguaglianza tra insiemi e tra liste richiedono l'unificazione. Infatti, dati due insiemi (o due liste) il vincolo di uguaglianza su di essi è soddisfatto se i due insiemi $"unificano"$, cioè se sono identici o se le variabili contenute in essi possono essere sostituite con oggetti tali da renderli identici. Quindi, in generale, il vincolo di uguaglianza tra insiemi o tra liste non si risolve in un semplice assegnamento, ma modifica entrambi gli insiemi (o entrambe le liste) coinvolti nel vincolo. Ad esempio, il vincolo di uguaglianza tra le liste $[X, 2]$ e $[1, Y]$ è soddisfatto con

$X = 1$ e $Y = 2$, mentre il vincolo di uguaglianza tra le liste $[X, 2]$ e $[Y, 1]$ non è soddisfacibile, in quanto per nessun valore di X e Y le due liste sono identiche. Sugli insiemi il discorso è analogo, ma bisogna tener presente che l'ordine degli elementi e le ripetizioni non contano; ad esempio, il vincolo $\{X, 2\} = \{Y, 1\}$ è soddisfatto con $X = 1$ e $Y = 2$, il vincolo $\{X, Y, 3\} = \{Z, 2\}$ è soddisfatta con Z uguale a 3 e con X e Y entrambe uguali a 2 o una uguale a 2 e l'altra uguale a 3, il vincolo $\{X, Y\} = \{1, 2\}$ è soddisfatto con $X = 1$ e $Y = 2$ oppure $X = 2$ e $Y = 1$, infine il vincolo $\{X, Y\} = \{Z, 2\}$ è soddisfatto con X, Y e Z uguali a 2 oppure con $X = Z$ e $Y = 2$ o ancora con $Y = Z$ e $X = 2$.

Analizziamo più in dettaglio l'unificazione tra liste e tra insiemi.

4.2.1 Unificazione tra liste

L'uguaglianza e la disuguaglianza sono gli unici vincoli definiti sulle liste. Date due liste `L1` e `L2` il vincolo `L1.eq(L2)` è soddisfatto se per ogni elemento della lista `L1` è soddisfatto il vincolo di uguaglianza tra quell'elemento e l'elemento che si trova nella stessa posizione nella lista `L2`. Quindi un vincolo di uguaglianza tra due `Lst` viene tradotto in vincoli di uguaglianza tra gli elementi delle due liste. Ad esempio il vincolo:

$[X, 2|R].eq([1, Z, 3, 4|S])$

comporta l'aggiunta al constraint store dei seguenti vincoli:

$X.eq(1)$
$Z.eq(2)$
$R.eq([3, 4|S])$

Il vincolo di disuguaglianza `L1.neq(L2)` è invece soddisfatto se le due liste `L1` e `L2` si differenziano almeno per un elemento. Ad esempio il vincolo

$[X, 2|R].neq([1, Z, 3, 4|S])$

è soddisfatto se lo è almeno uno dei seguenti vincoli:

$X.neq(1)$

$Z.neq(2)$

$R.neq([3,4|S])$

Consideriamo il seguente esempio:

Esempio 5

```
class UniLst1 {
 public static void main (String[] args)throws Fallimento{
  Lvar X = new Lvar();          // X: Lvar non inizializzata
  Lvar Y = new Lvar();          // Y: Lvar non inizializzata
  Lvar Z = new Lvar();          // Z: Lvar non inizializzata
  Lvar W = new Lvar();          // W: Lvar non inizializzata

  Lst l = new Lst(Lst.vuoto.insn(X).insn(2).insn(3).insn(Y));
  Lst m = new Lst(Lst.vuoto.insn(1).insn(Z).insn(W).insn(4));
                               // l = [X,2,3,Y], m = [1,Z,W,4]
  Solver.add(l.eq(m));         // vincolo di uguaglianza
  Solver.solve();
  return;
 }
}
```

La classe `UniLst1` crea quattro variabili logiche `X`, `Y`, `Z` e `W` tutte non inizializzate e le liste `l = [X,2,3,Y]` e `m = [1,Z,W,4]` sulle quali viene introdotto il vincolo di uguaglianza. Il metodo `solve` determina la soluzione: `X = 1`, `Y = 4`, `Z = 2` e `W = 3`, questi assegnamenti, infatti, rendono le due liste identiche. □

Consideriamo ora un esempio in cui compaiono liste non limitate. Le variabili `X`, `Z` e `W` sono definite come nell'Esempio 5:

Esempio 6

```
Lst L = new Lst();          // L: Lst non inizializzata
Lst M = new Lst();          // M: Lst non inizializzata

Lst l = new Lst(L.ins1(X).insn(2));
Lst m = new Lst(M.ins1(1).insn(Z).insn(W).insn(4).ins(5));
                            // l = [X,2|L], m = [1,Z,W,4,5|M]
Solver.add(l.eq(m));        // vincolo di uguaglianza
Solver.solve();
return;
```

In questo caso viene introdotto il vincolo di uguaglianza tra le liste `l = [X,2|L]` e `m = [1,Z,W,4,5|M]`. Il metodo `solve` trova la soluzione: `X = 1`, `Z = 2`, `L = [Lvar@2a9835,4,5|Set@62eec8]` dove `Lvar@2a9835` è l'identificatore usato dalla JVM per memorizzare la `Lvar` non inizializzata `W` e `Set@62eec8` è l'identificatore della lista non inizializzata `M`. □

4.2.2 Unificazione tra Set

Dati due insiemi s_1 e s_2 il vincolo $s_1.eq(s_2)$ è soddisfatto se ogni elemento di s_1 appartiene ad s_2 e viceversa.

Si consideri il seguente esempio nel quale le variabili `X`, `Y`, `Z` sono `Lvar` definite come nell'Esempio 5:

Esempio 7

```
Set s = new Set(Set.vuoto.ins(1).ins(Y));// s = {1,Y}
Set t = new Set(Set.vuoto.ins(X).ins(2).ins(Z));// t = {X,2,Z}
Solver.add(s.eq(t));                    // vincolo di uguaglianza
Solver.solve();
```

In questo esempio viene introdotto il vincolo $s = t$ con `s = {1,Y}` e `t = {X,2,Z}`. In modo non deterministico, viene trovata una possibile soluzione: `X = 1`, `Y = 2`, `Z = 2`. □

Nel prossimo esempio viene introdotto il vincolo di uguaglianza tra `Set` non limitati:

Esempio 8

```
class UniSet2 {
  public static void main (String[] args) {
    Lvar X = new Lvar();        // X: Lvar non inizializzata
    Lvar Y = new Lvar();        // Y: Lvar non inizializzata
    Lvar Z = new Lvar();        // Z: Lvar non inizializzata
    Set S = new Set();          // S: Set non inizializzato
    Set T = new Set();          // T: Set non inizializzato
    Set s = new Set(S.ins(1).ins(Y));        // s = {1,Y|S}
    Set t = new Set(T.ins(X).ins(2).ins(Z)); // t = {X,2,Z|T}
    Solver.add(s.eq(t));        // s = t
    Solver.solve();
    return;
  }
}
```

La classe `UniSet2` definisce il vincolo di uguaglianza $\{1, Y|S\} = \{X, 2, Z|T\}$. In modo non deterministico, viene trovata una possibile soluzione: `X = 1`, `Y` e `Z` non inizializzate, `S = {2|T}`. □

4.3 Vincoli di appartenenza e non appartenenza

I vincoli $l.in(s)$ e $l.nin(s)$ definiscono rispettivamente l'appartenenza e la non appartenenza di l ad s. l può essere un'istanza delle classi `Lvar`, `Lst` o `Set`, mentre s può essere un `Set` oppure una `Lvar` non inizializzata o inizializzata con un insieme; diversamente viene sollevata un eccezione `NotSetException` e la computazione termina.

Vediamo alcuni semplici esempi.

Esempio 9

```
class InNin1 {
   public static void main (String[] args){
     Lvar X = new Lvar();   // X: Lvar non inizializzata
     Lvar Y = new Lvar(2);  // Y: Lvar inizializzata con intero
     int[] ar = {1,2};
     Set s = new Set(ar);   // s = {1,2}

     Solver.add(X.in(s));   // vincolo di inclusione
     Solver.add(Y.in(s));   // vincolo di inclusione

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

La classe `InNin1` crea due variabili logiche `X` e `Y` ed un insieme `s`: `X` è una `Lvar` non inizializzata, `Y` è inizializzata con `2`, `s` è l'insieme `{1,2}`. I vincoli introdotti, in termini astratti sono: $X \in \{1,2\} \wedge Y \in \{1,2\}$. Questi due vincoli sono soddisfacibili. Al termine del metodo `solve` anche `X` è una `Lvar` inizializzata, il valore di `X` è `1`, in quanto è la prima soluzione, tra quelle possibili, trovata dal risolutore. Se avessimo introdotto anche il vincolo `X.neq(1)` il valore trovato per `X` sarebbe `2`. □

Se al posto di `Y.in(s)` introduciamo il vincolo `Y.nin(s)`, il problema non ha soluzioni. In questo caso viene generata un eccezione `NoSolutionFound` e la computazione termina.

Nei vincoli $l.in(s)$ e $l.nin(s)$, s può anche essere una `Lvar` non inizializzata oppure un `Set` non inizializzato o parzialmente specificato.

Esempio 10

```
class InNin2 {
   public static void main (String[] args) {
```

```
        Lvar X = new Lvar();   // X: Lvar non inizializzata
        Lvar Y = new Lvar();   // Y: Lvar non inizializzata
        Lvar Z = new Lvar(1);  // Z: Lvar inizializzata con intero

        Solver.add(Y.in(X));   // vincolo di appartenenza
        Solver.add(Z.in(X));   // vincolo di appartenenza
        Solver.add(Z.neq(Y));  // vincolo di disuguaglianza

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

In questo caso, i vincoli introdotti nel constraint store sono: $Y \in X \wedge 1 \in X \wedge Z \neq Y$. Al termine della `solve` la variabile logica `X` è una variabile inizializzata con il `Set` parzialmente specificato e non limitato $\{Y, 1|R\}$ dove con R indichiamo il resto, non noto, dell'insieme. □

Consideriamo il seguente esempio, nel quale `X`, `Y` e `Z` sono `Lvar` definite come nell'esempio 10 esempio:

Esempio 11

```
Set S = new Set(Set.vuoto.ins(X).ins(Y).ins(Z));  // S = {X,Y,1}
Set T = new Set(Set.vuoto.ins(1).ins(2).ins(Y));  // T = {1,2,Y}

Solver.add(T.in(S));                    // vincolo di appartenenza
Solver.solve();
```

In questo caso viene introdotto il vincolo $\{1, 2, Y\} \in \{X, Y, 1\}$, con `X` e `Y` variabili logiche non inizializzate. Il metodo `solve`, in questo caso, trova l'unica soluzione possibile: `X = {1,2,Y}` e quindi `S = {{1,2,Y},Y,1}`. La variabile `Y` resta non inizializzata. □

Si consideri ora la seguente classe:

Esempio 12

```
class InNin4 {
 public static void main (String[] args) {
   Lvar X = new Lvar();              // X: Lvar non inizializzata
   Lvar Y = new Lvar();              // Y: Lvar non inizializzata
   Set R = new Set();                // R: Set non inizializzato
   Set S = new Set(R.ins(1).ins(X)); // S = {1,X|R}
   Lst L = new Lst(Set.vuoto.ins1(2).insn(Y)); // L = [2,Y]

   Solver.add(L.in(S));              // vincolo di appartenenza
   Solver.solve();
   return;
 }
}
```

La classe `InNin4` definisce le variabili logiche `X` e `Y`, l'insiemi `S` e la lista `L`: `X` e `Y` sono `Lvar` non inizializzata, `S` è il `Set` {`1,X|R`} con `R` insieme non inizializzato. `L` è la lista `[2,Y]`. Su queste variabili è introdotto il vincolo `L.in(S)`, cioè in termini più astratti: $[2, Y] \in \{1, X|R\}$. Una possibile soluzione è quella ottenuta assegnando ad `X` la lista `L`, quindi `S` = {`1,[2,Y]|R`} con `Y` e `R` non inizializzate. □

Se aggiungiamo anche il vincolo `X.nin(R)`, questo non è soddisfatto dalla soluzione trovata in precedenza. In questo caso la soluzione è data da: `S` = {`1,X,[2,Y]|P`} con `X`, `Y` e `P` non inizializzate e `R` = {`[2,Y]|P`}.

Come si può vedere da questi esempi i vincoli `in` e `nin` sono molto diversi dalle operazioni `inl` e `ninl` definite sulle liste. I metodi `inl` e `ninl` sono dei semplici test che permettono di stabilire se un elemento appartiene o no alla lista, ma non modificano nè la lista su cui sono invocati, nè l'eventuale `Lvar` passata come parametro; i vincoli `in` e `nin` invece possono modificare, in modo non deterministico, le variabili logiche coinvolte nel vincolo.

4.4 Vincoli di confronto

In `JavaSet` sono definiti i seguenti vincoli di confronto:

$e_1.le(e_2)$ per la relazione di $''\leq''$

$e_1.lt(e_2)$ per la relazione di $''<''$

$e_1.ge(e_2)$ per la relazione di $''\geq''$

$e_1.gt(e_2)$ per la relazione di $''>''$

e_1 deve essere una `Lvar` e al momento della valutazione del vincolo il suo valore deve essere un intero o un `Integer`, e_2 può essere una espressione di tipo `int` o `Integer` oppure un'espressione di tipo `Lvar`, anch'essa al momento della valutazione del vincolo deve essere inizializzata con intero o `Integer`. Se non sono soddisfatte queste condizioni, viene sollevata un'eccezione `NotIntLvarException` e la computazione termina. I metodi `le`, `lt`, `ge` e `gt` non modificano la variabile logica su cui sono invocati o quella passata come parametro, ma servono solo per confrontare il valore di variabili logiche inizializzate con valori interi. Questi metodi vengono rivalutati in caso di backtracking. Ad esempio, supponiamo di aver introdotto e risolto il vincolo $X.gt(Y)$ con X e Y variabili logiche, se in seguito il valore delle due `Lvar` viene modificato a causa del backtracking, la relazione $X.gt(Y)$ viene testata di nuovo.

Consideriamo il seguente esempio nel quale vengono introdotti dei vincoli di confronto:

Esempio 13

```
class Conf1 {
   public static void main (String[] args) {
     Lvar X = new Lvar();        // X: Lvar non inizializzata
     Lvar Y = new Lvar();        // Y: Lvar non inizializzata
     Lvar Z = new Lvar();        // Z: Lvar non inizializzata
     int[] s_val = {2,5,8};      // s_val: array di interi
     Set s = new Set(s_val);     // s = {2,5,8}
```

```
        Solver.add(X.in(s).and(Y.in(s)).and(Z.in(s)));
                                          // vincoli di appartenenza
        Solver.add(X.gt(Y).and(Y.lt(Z))); // vincoli di confronto

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

La classe definisce tre variabili logiche non inizializzate: `X`, `Y` e `Z`. Queste variabili devono appartenere all'insieme {`2,5,8`} e devono soddisfare i vincoli `X > Y` e `Y < Z`, cioè in termini astratti i vincoli introdotti nel constraint store sono: $X \in \{2,5,8\} \wedge Y \in \{2,5,8\} \wedge Z \in \{2,5,8\} \wedge Y < X \wedge Y < Z$. Il metodo `solve` determina una possibile soluzione: `X = 5`, `Y = 2`, `Z = 5`.

Si noti che, nell'esempio 13, `X`, `Y` e `Z`, al momento della loro creazione, sono variabili non inizializzate; tuttavia i metodi `lt` e `gt` introdotti su di esse non generano una eccezione `NotIntLvarException`. Infatti quando i vincoli di confronto vengono valutati, le tre `Lvar` sono già state inizializzate a causa dei vincoli di appartenenza introdotti prima di quelli di confronto. Supponiamo invece di introdurre i vincoli di confronto prima di quelli di appartenenza, in questo caso viene sollevata l'eccezione in quanto, al momento della valutazione dei vincoli `X.gt(Y)` e `Y.lt(Z)`, le tre `Lvar` non sono ancora state inizializzate. Dunque è significativo l'ordine con cui vengono introdotti questi vincoli, non è così invece per gli altri vincoli insiemistici.

4.5 Vincoli aritmetici

Nella risoluzione di problemi con vincoli, capita spesso di dover introdurre e risolvere equazioni e disequazioni del tipo:

`L1 + L2 - L3 = L4`

`L1 + m - L2*L3` $\neq$ `n`

`L1/m + L2*n` $\geq$ `L3`

con L1, L2, L3 e L4 istanze di Lvar, n ed m di tipo intero.

La classe Lvar fornisce dei metodi che restituiscono una variabile logica il cui valore è il risultato di operazioni aritmetiche, tali metodi sono:

```
public Lvar sum(type e)
public Lvar sub(type e)
public Lvar mul(type e)
public Lvar div(type e)
```

e può essere un'espressione di tipo int o Integer, oppure un'espressione di tipo Lvar il cui valore, al momento della valutazione del vincolo deve essere un int o un Integer, se queste condizioni non sono soddisfatte, viene sollevata un'eccezione NotIntLvarException e la computazione termina. Tutti questi metodi restituiscono una Lvar, quindi possono essere concatenati, cioè alla Lvar restituita da uno di questi metodi può essere applicato un altro metodo.

Consideriamo ad esempio il seguente sistema:

$$\begin{cases} \texttt{L1 + L2 - L3 = L4} \\ \texttt{L1 + m - L2*L3} \neq \texttt{n} \end{cases}$$

queste relazioni possono essere introdotte come vincoli nel modo seguente:

```
Solver.add(L1.sum(L2).sub(L3).eq(L4));
Solver.add(L1.sum(m).sub(L2.mul(L3)).neq(L4));
```

Consideriamo ora il seguente esempio:

Esempio 14

```
class TestSumLvar {
   public static void main (String[] args) {
     Lvar x = new Lvar();        // x: Lvar non inizializzata
     Lvar y = new Lvar();        // y: Lvar non inizializzata
```

```
        Lvar z = new Lvar();         // z: Lvar non inizializzata
        int[] s_val = {1,2,3,4};
        Set s = new Set(s_val);      // s = {1,2,3}
        Solver.add(x.in(s));         // x.in({1,2,3})
        Solver.add(y.in(s));         // y.in({1,2,3})
        Solver.add(z.eq(x.sum(y))); // z.eq(x+y)
        Solver.solve();
        System.out.print("Lvar z = ");Lvar.output(z); // stampa z

        Solver.add(x.neq(1));        // x.neq(1)
        Solver.solve();
        System.out.print("Lvar z = ");Lvar.output(z); // stampa di z
        return;
      }
    }
```

L'output di questo programma è:

```
Lvar z = 2
Lvar z = 3
```

Quando viene invocato per la prima volta il metodo `solve`, i vincoli introdotti nel constraint store sono i seguenti: $x \in \{1,2,3\} \wedge y \in \{1,2,3\} \wedge z = x + y$. Quando viene dato il primo comando di stampa su `z` si ha che, ad `x` e ad `y` è stato assegnato, in modo non deterministico, il valore 1, quindi il valore il valore di `z` è 2; successivamente viene aggiunto il vincolo `x.neq(1)` e viene invocato di nuovo il metodo `solve`. Il valore assegnato in precedenza ad `x` non soddisfa questo nuovo vincolo, quindi grazie al backtracking viene ricercato un nuovo valore per `x` in modo che tutti i vincoli introdotti siano soddisfatti. Il nuovo valore assegnato a `x` è 2, quindi il valore di `z` viene ad essere 3. □

Come si vede dall'esempio, i metodi `sum`, `sub`, `mul` e `div` vengono rivalutati in caso di backtracking; cioè, la somma di due `Lvar` è una `Lvar` il cui valore viene

modificato se cambia il valore delle `Lvar` di cui è la somma; la stessa cosa vale per la differenza, il prodotto e il quoziente.

Si noti inoltre che, nell'esempio 14, `X` e `Y`, al momento della loro creazione, sono variabili non inizializzate; tuttavia il metodo `sum` non genera una eccezione `NotIntLvarException`. Infatti quando il vincolo `sum` viene valutato, le tre `Lvar` sono già state inizializzate a causa dei vincoli di appartenenza introdotti prima di quelli di confronto. Se invece avessimo introdotto il vincolo di somma prima di quelli di appartenenza, sarebbe stata sollevata l'eccezione in quanto, in quel caso, al momento della valutazione del vincolo `z.eq(x.sum(y))`, `x` e `y` non avrebbero avuto nessun valore. Dunque, come per i vincoli di confronto, è significativo l'ordine in cui i vincoli aritmetici vengono introdotti nel constraint store.

In effetti i vincoli di confronto e i vincoli aritmetici, si differenziano dagli altri vincoli. Questi vincoli in realtà sono dei test, in quanto possono essere applicati solo a variabili note. Per poterli trattare come gli altri vincoli, sarebbe necessario un risolutore adatto, in grado, ad esempio, di risolvere sistemi di equazioni e disequazioni in più incognite.

4.6 Vincoli di disgiunzione e non disgiunzione

I vincoli $s_1.disj(s_2)$ e $s_1.ndisj(s_2)$ definiscono rispettivamente l'operazione di disgiunzione e quella di non disgiunzione tra due insiemi $s_1 \parallel s_2$ e $s_1 \nparallel s_2$. Due insiemi sono disgiunti se non hanno nessun elemento comune, viceversa sono non disgiunti se almeno un elemento è comune ad entrambi; s_1 e s_2 possono essere `Lvar` non inizializzate o inizializzate con insieme, oppure istanze della classe `Set`. In caso contrario viene sollevata un eccezione `NotSetException` e la computazione termina.

Ad esempio, gli insiemi $\{1, 2, 3\}$ e $\{4, 5\}$ sono disgiunti, mentre gli insiemi $\{1, 2, 3\}$ e $\{2, 4, 5\}$ sono non disgiunti. Gli insiemi coinvolti nei vincoli di disgiunzione e non disgiunzione possono essere anche insiemi parzialmente specificati, limitati o non limitati. Ad esempio gli insiemi $\{1, X, 3|R_1\}$ e $\{Y, Z|R_2\}$, con X, Y e Z `Lvar` e R_1 ed R_2 `Set` non inizializzati, sono disgiunti se $X \notin R_2 \wedge X \neq Y \wedge$

$X \neq Z \wedge 1 \notin R_2 \wedge 3 \notin R_2 \wedge Y \neq 1 \wedge Y \neq 3 \wedge Z \neq 1 \wedge Z \neq 3 \wedge Y \notin R_1 \wedge Z \notin R_2 \wedge R_1 \parallel R_2$, cioè se X non appartiene ad R_2, X è diversa da Y e da Z, 1 e 3 non appartengono a R_2, Y e Z sono diversi da 1 e 3 e non appartengono a R_1, ed R_1 è disgiunto da R_2.

Consideriamo il seguente esempio:

Esempio 15

```
class EsemDisj1 {
 public static void main (String[] args)throws Fallimento {
   Lvar X = new Lvar();            // X: Lvar non inizializzata
   Lvar Y = new Lvar();            // Z: Lvar non inizializzata
   Set R1 = new Set();             // R1: Set non inizializzato
   Set R2 = new Set();             // R2: Set non inizializzato
   Set r = new Set(Set.vuoto.ins(1).ins(2));       // r = {1,2}

   Solver.add(X.in(r).and(Y.in(r))); // vincoli di appartenenza
   Solver.add(R1.ins(X).ins(1).disj(R2.ins(Y).ins(2)));
                                        // vincolo di disgiunzione
   Solver.solve();
   return;
 }
}
```

Nel'esempio 15 sono definiti i vincoli $X \in \{1,2\} \wedge Y \in \{1,2\} \wedge \{X,1|R1\} \parallel \{Y,2|R2\}$ dove `X` e `Y` sono `Lvar` non inizializzate e `R1` ed `R2` sono `Set` non inizializzati, di conseguenza {`X,1|R1`} e {`Y,2|R2`} sono entrambi `Set` parzialmente definiti non limitati. Il metodo `solve` trova la soluzione: `X` inizializzata con `1` e `Y` inizializzata con `2`. Gli insiemi `R1` e `R2` restano non inizializzati, ma su di essi è stato aggiunto il vincolo di disgiunzione, infatti affinchè {`1|R1`} e {`2|R2`} siano disgiunti è necessario che lo siano anche i rispettivi resti. □

Consideriamo ora lo stesso esempio ma con l'aggiunta di alcuni vincoli:

Esempio 16

```
Set t = new Set(Set.vuoto.ins(3).ins(4)); // t = {3,4}
Solver.add(R1.ndisj(t).and(R2.ndisj(t))); // vincoli di non disg.
Solver.solve();
```

Sono stati aggiunti i vincoli `R1.ndisj(t)` e `R2.ndisj(t)` con `t = {3,4}`. Il metodo `solve` trova la soluzione: `X` inizializzata con `1`, `Y` inizializzata con `2`, `R1 = {3|T1}` e `R2 = {4|T2}` con `T1` e `T2` insiemi non inizializzati. Durante la risoluzione è stato aggiunto il vincolo `T1.disj(T2)`, infatti se così non fosse, in seguito, `T1` e `T2` potrebbero essere inizializzati con insiemi non disgiunti, e quindi i vincoli non sarebbero soddisfatti. □

4.7 Vincoli di unione e non unione

Il vincolo $s_1.union(s_2, s_3)$ definisce $s_1 = s_2 \cup s_3$, mentre il vincolo $s_1.nunion(s_2, s_3)$ definisce $s_1 \neq s_2 \cup s_3$; s_1, s_2 e s_3 possono essere `Lvar` non inizializzate o inizializzate con insieme, oppure istanze della classe `Set`. In caso contrario viene sollevata un'eccezione `NotSetException` e la computazione termina. I `Set` coinvolti nell'unione o nella non unione possono essere di qualunque tipo, cioè possono essere insiemi inizializzati o non inizializzati, completamente o parzialmente specificati, limitati o non limitati.

Vediamo alcuni esempi:

Esempio 17

```
class Union1 {
   public static void main (String[] args){
     Set r = new Set(Set.vuoto.ins(1).ins(2));    // r = {1,2}
     Set s = new Set(Set.vuoto.ins(3).ins(4));    // s = {3,4}
     Set T = new Set();            // T: Set non inizializzato
```

```
      Solver.add(T.union(r,s));      // vincolo di unione
      Solver.solve();
      return;
    }
  }
```

In questo esempio viene introdotto e risolto il vincolo $T = r \cup s$ con `r = {1,2}`, `s = {3,4}` e `T` insieme non inizializzato; al termine del metodo `solve` l'insieme `T` è inizializzato, il suo valore è `{1,2,3,4}`. □

Consideriamo un altro esempio:

Esempio 18

```
class Union2 {
  public static void main (String[] args) {
     Set r = new Set(Set.vuoto.ins(1).ins(2)); // r = {1,2}
     Set t = new Set(r.ins(3).ins(4));         // t = {1,2,3,4}
     Set S = new Set();            // S = Set non inizializzato
     Solver.add(t.union(r,S));     // vincolo di unione
     Solver.solve();
     return;
  }
}
```

In questo caso viene introdotto e risolto il vincolo `t.union(r,S)` con `r = {1,2}`, `S` insieme non inizializzato e `t = {1,2,3,4}`. Al termine del metodo `solve` l'insieme `S` è inizializzato, il suo valore è `{3,4}`. Questa soluzione non è l'unica possibile, infatti se aggiungiamo il vincolo `S.neq(s1)` con `s1 = {3,4}`, la soluzione trovata in precedenza non soddisfa questo nuovo vincolo, quindi il risolutore cerca un'altra soluzione che soddisfi entrambi i vincoli e il nuovo valore trovato per l'insieme `S` è `{2,3,4}`. □

Consideriamo ora alcuni esempi sull'uso della `union` nel caso di `Set` parzialmente specificati:

Esempio 19

```
class Union3 {
  public static void main (String[] args) throws Fallimento {
    Lvar X = new Lvar();              // X = Lvar non inizializzata
    Lvar Y = new Lvar();              // Y = Lvar non inizializzata
    Lvar Z = new Lvar();              // Z = Lvar non inizializzata
    Lvar W = new Lvar();              // W = Lvar non inizializzata
    Set r = new Set(Set.vuoto.ins(1).ins(X).ins(Y));
                                      // Set r = {1,X,Y}
    Set s = new Set(Set.vuoto.ins(2).ins(X).ins(Z));
                                      // Set s = {2,X,Z}
    Set t = new Set(Set.vuoto.ins(2).ins(3).ins(W).ins(4));
                                      // Set t = {2,3,W,4}
    Solver.add(t.union(r,s));         // vincolo di unione
    Solver.solve();

  ///////////////     istruzioni per la stampa     ///////////////

    System.out.print("\nSet r = ");Set.output(r);
    System.out.print("\nSet s = ");Set.output(s);
    System.out.print("\nSet t = ");Set.output(t);
    System.out.print("\nLvar X = ");Lvar.output(X);
    System.out.print("\nLvar Y = ");Lvar.output(Y);
    System.out.print("\nLvar Z = ");Lvar.output(Z);
    System.out.print("\nLvar W = ");Lvar.output(W);
    return;
  }
}
```

In questo esempio viene introdotto il vincolo $\{2,3,W,4\} = \{1,X,Y\} \cup \{2,X,Z\}$ con `X`, `Y`, `Z` e `W` variabili logiche non inizializzate. L'output di questo programma è il seguente:

```
Set r = {4,2,1}
Set s = {3,2,2}
Set t = {4,1,3,2}
Lvar X = 2
Lvar Y = 4
Lvar Z = 3
Lvar W = 1
```

□

Diamo un ultimo esempio sull'uso dei vincoli di unione e non unione:

Esempio 20

```
class UnionNunion {
 public static void main (String[] args) throws Fallimento {
  Lvar X = new Lvar();              // X: Lvar non inizializzata
  Lvar Y = new Lvar();              // Y: Lvar non inizializzata
  Lvar Z = new Lvar();              // Z: Lvar non inizializzata
  Lvar W = new Lvar();              // W: Lvar non inizializzata
  Set R = new Set();                // R: Set non inizializzato
  Set S = new Set();                // S:  Set non inizializzato
  Set r = new Set(R.ins(1).ins(X).ins(Y)); // r = {1,X,Y|R}
  Set s = new Set(S.ins(2).ins(X).ins(Z)); // s = {2,X,Z|S}
  Set r1 = new Set(Set.vuoto.ins(1).ins(X).ins(Y)); // r1 = {1,X,Y}
  Set s1 = new Set(Set.vuoto.ins(2).ins(X).ins(Z)); // s1 = {2,X,Z}
  Set t = new Set(Set.vuoto.ins(2).ins(3).ins(W).ins(4));
                                    // t = {2,3,W,4}
  Solver.add(t.union(r,s));         // vincolo di unione
  Solver.add(t.nunion(r1,s1));      // vincolo di non unione
  Solver.solve();
   ...                              // istruzioni per la stampa
  return;
 }
}
```

In questo esempio vengono introdotti i vincoli:

$$\{2,3,W,4\} = \{1,X,Y|R\} \cup \{2,X,Z|S\} \wedge \{2,3,W,4\} \neq \{1,X,Y\} \cup \{2,X,Z\}$$

con X, Y, Z e W variabili logiche non inizializzate e R ed S insiemi non inizializzati. In questo caso, quindi, prima della risoluzione dei vincoli, r e t sono Set parzialmente specificati non limitati. Questo programma ha il seguente output:

```
Set r = {1,2,4,3}
Set s = {2,2,2}
Set t = {2,3,1,4}
Set r1 = {1,2,4}
Set s1 = {2,2,2}
Lvar X = 2
Lvar Y = 4
Lvar Z = 2
Lvar W = 1
Set R = {3}
Set S = {}
```

□

4.8 Vincoli di inclusione e non inclusione

I vincoli $s_1.subset(s_2)$ e $s_1.nsubset(s_2)$ definiscono rispettivamente $s_1 \subseteq s_2$ e $s_1 \not\subseteq s_2$; s_1 e s_2 possono essere Lvar non inizializzate o inizializzate con insieme, oppure possono essere istanze della classe Set, in caso contrario viene sollevata un eccezione NotSetException e la computazione termina.

Vediamo alcuni esempi:

Esempio 21

```
class Subset1 {
 public static void main (String[] args) {
  int[] a = {1,2,3};
  Set s = new Set(a);          // s = {1,2,3}
  Set r = Set.mkSet(2);        // r = Set di due Lvar
  Solver.add(r.subset(s));     // vincolo di inclusione
  Solver.solve();
```

```
    return;
   }
  }
```

In questo esempio sono definiti gli insiemi `s = {1,2,3}` e l'insieme `r` costituito da due `Lvar` non inizializzate. Su questi due insiemi è definito il vincolo: `r.subset(s)`. La soluzione trovata dal metodo `solve` è `r = {3}`, cioè le due `Lvar` dell'insieme `r` sono entrambe uguali a `3`. □

Consideriamo un altro esempio:

Esempio 22

```
class Subset2 {
  public static void main (String[] args){
   Lvar X = new Lvar();          // X: Lvar non inizializzata
   Set R = new Set();            // R: Set non inizializzato
   Set s = new Set(R.add(1));    // s = {1|R}
   Set t = new Set(Set.vuoto.add(3).add(2).add(X)); // t = {2,3,X}
   Solver.add(t.subset(s));      // vincolo di inclusione
   Solver.solve();
   return;
  }
}
```

La classe `Subset2` definisce un vincolo di inclusione tra gli insiemi `t = {3,2,X}` e `s = {1|R}` con `X` variabili logica non inizializzata e `R` insieme non inizializzato. La soluzione trovata dalla `solve` è la seguente: `s = {1,2,3,X|S}`, `t = {2,3,X}`, `R = {2,3,X|S}` con `X` `Lvar` non inizializzata e `S` `Set` non inizializzato. □

4.9 Vincoli di intersezione e non intersezione

Il vincolo $s_1.inters(s_2, s_3)$ definisce $s_1 = s_2 \cap s_3$, mentre il vincolo $s_1.ninters(s_2, s_3)$ definisce $s_1 \neq s_2 \cap s_3$. I parametri passati a questi due metodi possono essere `Lvar`

non inizializzate o inizializzate con insieme, oppure istanze della classe `Set`. In caso contrario viene sollevata un eccezione `NotSetException` e la computazione termina. I `Set` coinvolti nell'intersezione o nella non intersezione possono essere di qualunque tipo, cioè possono essere insiemi inizializzati o non inizializzati, completamente o parzialmente specificati, limitati o non limitati.

Vediamo alcuni esempi:

Esempio 23

```
class Inters0 {
  public static void main (String[] args) {
    Set r = new Set(Set.vuoto.ins(1).ins(2));     // r = {1,2}
    Set s = new Set(Set.vuoto.ins(3).ins(4));     // s = {3,2}
    Set T = new Set();             // T: Set non inizializzato
    Solver.add(T.inters(r,s));     // vincolo di intersezione
    Solver.solve();
    return;
  }
}
```

In questo esempio viene introdotto e risolto il vincolo $T = r \cap s$ con `r =` $\{$`1,2`$\}$, `s =` $\{$`3,2`$\}$ e `T` insieme non inizializzato; al termine del metodo `solve` l'insieme `T` è inizializzato, il suo valore è $\{$`2`$\}$. □

Consideriamo un altro esempio:

Esempio 24

```
class Inters1 {
  public static void main (String[] args) {
    Set r = new Set(Set.vuoto.ins(1).ins(2)); // r = {1,2,3}
    Set t = new Set(r.ins(3).ins(4));         // t = {1,2}
    Set S = new Set();           // S: Set non inizializzato
    Solver.add(t.inters(r,S));   // vincolo di intersezione
```

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

In questo caso viene introdotto e risolto il vincolo $t = r \cap S$ con `r = {1,2,3}`, `S` insieme non inizializzato e `t = {1,2}`. Al termine del metodo `solve` l'insieme `S` è inizializzato, il suo valore è `{1,2|R}` con `R` insieme non inizializzato. □

Consideriamo ora un esempio sull'uso della `inters` e della `ninters` nel caso di `Set` parzialmente specificati:

Esempio 25

```
class Inters2 {
 public static void main (String[] args){
  Lvar X = new Lvar();          // X: Lvar non inizializzata
  Lvar Y = new Lvar();          // Y: Lvar non inizializzata
  Lvar Z = new Lvar();          // Z: Lvar non inizializzata
  Lvar W = new Lvar();          // W: Lvar non inizializzata
  Set r = new Set(Set.vuoto.ins(1).ins(X).ins(Y));
                                              // Set r = {1,X,Y}
  Set s = new Set(Set.vuoto.ins(2).ins(Z)); // Set s = {2,Z}
  Set t = new Set(Set.vuoto.ins(3).ins(W)); // Set t = {3,W}
  Set t1 = new Set(Set.vuoto.ins(3));       // Set t1 = {3}
  Solver.add(t.inters(r,s));    // vincolo di intersezione
  Solver.add(t1.ninters(r,s));  // vincolo di non intersezione
  Solver.solve();

///////////////    istruzioni per la stampa    ///////////////

  System.out.print("\nSet r = ");Set.output(r);
  System.out.print("\nSet s = ");Set.output(s);
  System.out.print("\nSet t = ");Set.output(t);
  System.out.print("\nLvar X = ");Lvar.output(X);
```

```
      System.out.print("\nLvar Y = ");Lvar.output(Y);
      System.out.print("\nLvar Z = ");Lvar.output(Z);
      System.out.print("\nLvar W = ");Lvar.output(W);
      return;
    }
  }
```

In questo esempio vengono introdotti i vincoli $\{3, W\} = \{1, X, Y\} \cap \{2, Z\}$ e $\{3\} \neq \{1, X, Y\} \cap \{2, Z\}$, con `X`, `Y`, `Z` e `W` variabili logiche non inizializzate. L'output di questo programma è il seguente:

```
Set r = {1,2,3}
Set s = {2,3}
Set t = {3,2}
Lvar X = 3
Lvar Y = 2
Lvar Z = 3
Lvar W = 2
```

□

4.10 Vincoli di differenza e non differenza

Il vincolo $s_1.differ(s_2, s_3)$ definisce l'operazione di differenza tra due insiemi $s_1 = s_2 \setminus s_3$, mentre il vincolo $s_1.ndiffer(s_2, s_3)$ definisce l'operazione $s_1 \neq s_2 \setminus s_3$; s_1, s_2 e s_3 possono essere `Lvar` non inizializzate o inizializzate con insieme, oppure istanze della classe `Set`. In caso contrario viene sollevata un eccezione `NotSetException` e la computazione termina. Gli insiemi coinvolti nei vincoli di differenza e non differenza possono essere di qualunque tipo, cioè possono essere insiemi inizializzati o non inizializzati, completamente o parzialmente specificati, limitati o non limitati.

Vediamo alcuni esempi:

Esempio 26

```
class Difference1 {
   public static void main (String[] args) {
     Set r = new Set(Set.vuoto.ins(1).ins(3).ins(4));
                                                  // r = {1,3,4}
     Set s = new Set(Set.vuoto.ins(2).ins(1)); // s = {2,1}
     Set T = new Set();           // T: Set non inizializzato
     Solver.add(T.differ(r,s));   // vincolo di differenza
     Solver.solve();
     return;
   }
}
```

In questo esempio viene introdotto e risolto il vincolo $T = r \setminus s$ con `r` = {1,3,4}, `s` = {2,1} e `T` insieme non inizializzato; al termine del metodo `solve` l'insieme `T` è inizializzato, il suo valore è {3,4} cioè è l'insieme ottenuto dalla differenza insiemistica tra `r` e `s`. □

Consideriamo un altro esempio:

Esempio 27

```
class Difference2 {
  public static void main (String[] args) {
    Set r = new Set(Set.vuoto.ins(1).ins(2).ins(3).ins(4));
                                                  // r = {1,2,3,4}
    Set t = new Set(Set.vuoto.ins(3).ins(4)); // t = {3,4}
    Set S = new Set();             // S: Set non inizializzato
    Solver.add(t.differ(r,S));     // vincolo di differenza
    Solver.solve();
    return;
  }
}
```

In questo caso viene introdotto e risolto il vincolo `t.differ(r,S)` con `r` = {1,2,3,4},

`S` insieme non inizializzato e `t = {3,4}`. Al termine del metodo `solve` l'insieme `S` è inizializzato, il suo valore è `{1,2|R}` con `R` insieme non inizializzato. □

Consideriamo ora alcuni esempi sull'uso della `differ` nel caso di `Set` parzialmente specificati:

Esempio 28

```
class Difference3 {
  public static void main (String[] args) throws Fallimento {
    Lvar X = new Lvar();           // X = Lvar non inizializzata
    Lvar Y = new Lvar();           // Y = Lvar non inizializzata
    Set r = new Set(Set.vuoto.ins(2).ins(X).ins(Z));
                                                 // Set r = {2,X,Y}
    Set s = new Set(Set.vuoto.ins(1).ins(X)); // Set s = {1,X}
    Set t = new Set(Set.vuoto.ins(2));        // Set t = {2}
    Solver.add(t.differ(r,s));     // vincolo di differenza
    Solver.solve();
    return;
  }
}
```

In questo esempio viene introdotto il vincolo di differenza tra gli insiemi `r = {2,X,Y}`, `s = {1,X}` e `t = {2}` con `X` e `Y` variabili logiche non inizializzate. Questo problema ammette la soluzione: `Y = 1`, `X` non inizializzata. □

4.11 Il vincolo less

Il vincolo $S_1.less(L, S_2)$, dove S_1 e S_2 sono insiemi e L è una variabile logica, realizza l'estrazione di L dall'insieme S_1, infatti l'insieme S_2 è ottenuto da S_1 eliminando tutte le occorrenze di L; S_1 e S_2 possono essere insiemi qualunque, cioè inizializzati o no, parzialmente o completamente specificati, limitati o no. Anche L può essere inizializzata o no.

Se L non è presente in S_1, o S_1 è vuoto, la `less` fallisce; in questo il vincolo `less` è diverso dal vincolo `differ`, infatti $S_2.differ(S_1, \{L\})$, nel caso in cui $\{L\}$ e S_1 siano disgiunti non fallisce, ma equivale al vincolo $S_1.eq(S_2)$.

Consideriamo il seguente esempio:

Esempio 29

```
class TestLess1 {
  public static void main (String[] args) {
    int[] ar = {1,2,3};
    Lvar x = new Lvar(1);     // x: Lvar inizializzata con intero
    Set s = new Set(Set.vuoto.ins(x).insAll(ar)); // s = {1,2,3,x}
    Set T = new Set();        // t: Set non inizializzato

    Solver.add(s.less(x,T));
    Solver.solve();
    System.out.print("\nLvar x = ");Lvar.output(x);
    System.out.print("\nSet s = ");Set.output(s);
    System.out.print("\nSet T = ");Set.output(T);
    return;
  }
 }
```

In questo esempio vengono definiti la variabile logica `x`, inizializzata con l'intero `1`, l'insieme `s = {1,2,3,x}` e l'insieme `T` non inizializzato, poi viene introdotto il vincolo `s.less(x,T)`.

L'output del programma è il seguente:

```
Lvar x = 1
Set s = {1,2,3,1}
Set T = {2,3}
```

L'insieme `T` è l'insieme ottenuto eliminando da `s` le occorrenze di `x`. □

Consideriamo un altro esempio:

Esempio 30

```
class TestLess2 {
  public static void main (String[] args) {
    int[] ar = {1,2,3};
    Lvar x = new Lvar(4);   // x: Lvar inizializzata con intero
    Set S = new Set();      // S: Set non inizializzato
    Set t = new Set(ar);    // t = {1,2,3}

    Solver.add(S.less(x,t));
    Solver.solve();
    System.out.print("\nLvar x = ");Lvar.output(x);
    System.out.print("\nSet S = ");Set.output(S);
    System.out.print("\nSet t = ");Set.output(t);
    return;
  }
}
```

In questo esempio vengono definiti la variabile logica `x`, inizializzata con l'intero `4`, l'insieme `S` non inizializzato e l'insieme `t = {1,2,3}`, poi viene introdotto il vincolo `S.less(x,t)`.

L'output del programma è il seguente:

```
Lvar x = 4
Set S = {4,1,2,3}
Set t = {1,2,3}
```

L'insieme `S` è l'insieme ottenuto aggiungendo a `t` la variabile `x`. □

Consideriamo un ultimo esempio:

Esempio 31

```
class TestLess3{
  public static void main (String[] args)
  throws IOException,Fallimento {
    Lvar X = new Lvar();    // X: Lvar non inizializzata
    Lvar Y = new Lvar();    // Y: Lvar non inizializzata
    Lvar Z = new Lvar();    // Z: Lvar non inizializzata
    int[] ar = {1,2,3};
    Set r = new Set(ar);    // s = {1,2,3}
    Set S = new Set();      // S: Set non inizializzato
    Set T = new Set();      // T: Set non inizializzato
    Set U = new Set();      // U: Set non inizializzato

    Set.less(r,X,S);
    Set.less(S,Y,T);
    Set.less(T,Z,U);
    Solver.solve();

    System.out.print("\nLvar X = ");Lvar.output(X);
    System.out.print("\nLvar Y = ");Lvar.output(Y);
    System.out.print("\nLvar Z = ");Lvar.output(Z);
    System.out.print("\nSet S = ");Set.output(S);
    System.out.print("\nSet T = ");Set.output(T);
    System.out.print("\nSet U = ");Set.output(U);
    return;
  }
 }
```

La classe `TestLess3` introduce tre vincoli: `r.less(X,S)`,`S.less(Y,T)` e `T.less(Z,U)`, con `X`, `Y` e `Z` variabili non inizializzate, `S`, `T` e `U` insiemi non inizializzati e `r = {1,2,3}`.

L'output è il seguente:

```
Lvar X = 1
Lvar Y = 2
Lvar Z = 3
Set S = {2,3}
Set T = {3}
Set U = {}
```

□

4.12 Inserimento dei vincoli

Il metodo `add` permette di memorizzare i vincoli nel *"constraint store"*, cioè nel *"contenitore"* dei vincoli. Quindi, per introdurre un vincolo, bisogna definirlo e memorizzarlo mediante la `add`. Il parametro passato alla `add` può essere un singolo vincolo oppure una congiunzione di vincoli. Ad esempio le istruzioni $Solver.add(v)$ e $Solver.add(v_1.and(v_2)....and(v_n))$ aggiungono allo store, rispettivamente, il vincolo v e i vincoli $v_1 ... v_n$.

Aggiungere allo store una congiunzione di vincoli è del tutto equivalente ad aggiungere un vincolo per volta, quindi l'istruzione $Solver.add(v_1.and(v_2)....and(v_n))$ è equivalente alle istruzioni: $Solver.add(v_1) ... Solver.add(v_n)$.

Si noti che il metodo statico `add` della classe Solver è del tutto simile al metodo statico `post`, usato per lo stesso scopo, in `JSolver` [5] (cfr. sezione 2.1.3 pag. 11).

Il metodo `add` non è l'unico metodo che realizza l'introduzione dei vincoli, infatti la classe `Solver` prevede altri due metodi statici che aggiungono vincoli al constraint store: il metodo `AllDifferent` e il metodo `forall`.

4.12.1 Il metodo *allDifferent*

L'espressione $Solver.allDifferent(S)$ serve per introdurre i vincoli di disuguaglianza tra gli elementi di un insieme S. In termini astratti quindi $Solver.allDifferent(S)$ equivale all'espressione $\forall s_1, s_2 \in S, s_1 \neq s_2$.

S può essere una `Lvar` non inizializzata o inizializzata con insieme, oppure un'espressione di tipo `Set`. In caso contrario viene sollevata un'eccezione di tipo `NotSetException` e la computazione termina.

S può essere un insieme di qualunque tipo, cioè può essere un insieme inizializzato o non inizializzato, completamente o parzialmente specificato, limitato o non limitato.

Consideriamo il seguente esempio:

Esempio 32

```
class AllDifferent1 {
  public static void main (String[] args)
  throws Fallimento{
    Lvar X = new Lvar();      // X: Lvar non inizializzata
    Lvar Y = new Lvar();      // Y: Lvar non inizializzata
    Lvar Z = new Lvar();      // Z: Lvar non inizializzata
    Set r = new Set(Set.vuoto.ins(X).ins(Y).ins(Z)); // r = {X,Y,Z}
    Set s = new Set(Set.vuoto.ins(1).ins(2).ins(3)); // s = {1,2,3}

    Solver.allDifferent(r);
    Solver.add(X.in(s).and(Y.in(s)).and(Z.in(s)));
                              // vincoli di appartenenza
    Solver.solve();
    return;
  }
}
```

In questo esempio viene invocato il metodo `allDifference` sull'insieme `r` = {`X`,`Y`,`Z`}, con `X`, `Y` e `Z` variabili logiche non inizializzate, questo equivale ad introdurre i vincoli $X \neq Y \wedge X \neq Z \wedge Y \neq Z$. Inoltre `X`, `Y` e `Z` devono appartenere all'insieme `s` = {`1,2,3`} cioè devono essere soddisfatti anche i vincoli $X \in \{1,2,3\} \wedge Y \in \{1,2,3\} \wedge Z \in \{1,2,3\}$. I vincoli sono soddisfacibili, quindi il metodo `solve` trova una delle possibili soluzioni: `X = 1`, `Y = 2`, `Z = 3`.

Se invece, ad esempio, `s` fosse uguale all'insieme {`1,2`} i vincoli non sarebbero soddisfacibili. In questo caso verrebbe sollevata un'eccezione `NoSolutionFound` e la computazione si interromperebbe. □

4.12.2 Il metodo *forall*

La classe `Solver` fornisce il metodo statico `forall` che realizza la struttura di controllo **for**, cioè permette di applicare uno o più vincoli a ciascun elemento di un insieme. L'espressione $Solver.forall(X, S, v_X)$ in termini astratti equivale alla

espressione $\forall X \in S$, v_X , dove X è una `Lvar` non inizializzata, S un insieme e v_X un vincolo o una congiunzione di vincoli; v_X può essere un vincolo di qualunque tipo, naturalmente in v_X deve comparire la variabile X, anche se, in caso contrario, non si incorre in nessun errore.

Consideriamo il seguente esempio:

Esempio 33

```
class TestForall1 {
  public static void main (String[] args)
  throws Fallimento {
    Lvar X = new Lvar();     // X: Lvar non inizializzata
    Lvar Y = new Lvar();     // Y: Lvar non inizializzata
    Lvar Z = new Lvar();     // Z: Lvar non inizializzata
    int[] ar = {1,2};
    Set s = new Set(ar);     // s = {1,2}
    Set t = new Set(Set.vuoto.ins(Z).ins(Y).ins(X));
                             // t = {X,Y}
    Lvar L = new Lvar();     // L: Lvar non inizializzata

    Solver.forall(L,t,L.in(s));
    Solver.solve();
    return;
  }
}
```

In questo esempio i vincoli introdotti dal metodo `forall`, in termini astratti, sono: $\forall L \in \{X, Y, Z\}, L \in \{1, 2\}$, cioè vengono introdotti i vincoli $X \in \{1, 2\} \wedge Y \in \{1, 2\} \wedge Z \in \{1, 2\}$. I vincoli sono soddisfacibili, quindi il metodo `solve` determina una possibile soluzione: `X = 1`, `Y = 2`, `Z = 2`. □

Consideriamo l'esempio seguente, in cui `X`, `Y`, `Z` e `t` sono definiti come nell'esempio precedente:

Esempio 34

```
int[] s_val = {1,2,3,4,5,6};
int[] u_val = {2,4,7,5,6};
Set s = new Set(s_val);          // s = {1,2,3,4,5,6}
Set u = new Set(u_val);          // u = {2,4,7,5,6}
Lvar L = new Lvar();             // L: Lvar non inizializzata
Solver.forall(L,t,L.in(s).and(L.in(u)));
Solver.solve();
```

In questo esempio il metodo `forall` introduce una congiunzione di vincoli per ogni elemento dell'insieme {`X,Y,Z`}. In termini astratti, i vincoli introdotti sono: $\forall L \in \{X, Y, Z\}, L \in \{1, 2, 3, 4, 5, 6\} \wedge L \in \{2, 4, 7, 5, 6\}$. Il metodo `solve` trova una possibile soluzione: `X = 2`, `Y = 4`, `Z = 5`. □

Consideriamo il seguente esempio, in cui `X`, `Y`, `Z` e `t` sono definiti come negli esempi precedenti:

Esempio 35

```
int[] s_val = {1,2,3};
Set s = new Set(ar);
Set R = new Set();                // R: Set non inizializzato
Lvar L = new Lvar();              // L: Lvar non inizializzata
Set l = new Set(Set.vuoto.ins(L).ins(2).ins(1)); //l = {1,2,L}

Solver.forall(L,t,L.in(s).and(Lset.in(R)));
Solver.solve();

System.out.print("\nSet R = ");Set.output(R);
System.out.print("\nLvar X = ");Lvar.output(X);
System.out.print("\nLvar Y = ");Lvar.output(Y);
System.out.print("\nLvar Z = ");Lvar.output(Z);
return;
```

In termini astratti, i vincoli introdotti in questo caso sono: $\forall L \in \{X, Y, Z\}, L \in \{1, 2, 3\} \wedge \{1, 2, L\} \in R$ con R insieme non inizializzato. L'output del programma è il seguente.

```
Set R = {{2,1,1},{2,1,3}|Set@1bd03e}
Lvar X = 1
Lvar Y = 2
Lvar Z = 3
```

dove `Set@1bd03e` è l'identificatore usato dalla JVM per identificare l'oggetto che rappresenta il resto non noto di `R`. □

Consideriamo ora un esempio in cui il metodo `forall` viene utilizzato per calcolare il massimo di un insieme:

Esempio 36

```
class Max {
  public static void max(Set s)throws Fallimento {
    Lvar m = new Lvar();    // m: Lvar non inizializzata
    Solver.add(m.in(s));    // vincolo di appartenenza
    Lvar X = new Lvar();    // X: Lvar non inizializzata
    Solver.forall(X,s,X.le(m));
    return;
  }

  public static void main (String[] args) {
    int[] a = {1,6,2,8,5};
    Set s = new Set(a);
    max(s);
    Solver.solve();
  }
}
```

In questo esempio, in termini astratti, i vincoli introdotti sono:

$$m \in \{1,6,2,8,5\} \wedge \forall X \in \{1,6,2,8,5\},\ X \leq m$$

con `m` variabile logica non inizializzata. L'unica soluzione possibile è `m = 8`.

Analogamente la ricerca del minimo di un insieme si può realizzare sostituendo `Solver.forall(X,s,X.le(m))` con `Solver.forall(X,s,X.ge(m))`. Nel nostro esempio la soluzione sarebbe `m = 1`. □

4.13 Il risolutore dei vincoli

Come si è detto le tecniche di risoluzione adottato in JavaSet sono le stesse utilizzate in SINGLETON [16] e definite in CLP($\mathcal{SET}$) [8]. Il risolutore del linguaggio CLP($\mathcal{SET}$) cerca di ridurre ciascun vincolo presente nel constraint store in una forma semplificata, detta *forma risolta* che può essere testata facilmente per determinare se è soddisfacibile. Il successo di questo processo di riduzione permette di concludere che il constraint store, nella sua forma originale, è soddisfacibile. Viceversa un fallimento implica l'insoddisfacibilità dei vincoli.

Un vincolo, appartenente ad un insieme C di vincoli dati, è in forma risolta se è in una delle seguenti forme:

- $X = t$, e X non è contenuta né in t, né nel negli altri vincoli di C;
- $X \neq t$, e non c'è nessuna occorrenza di X in t;
- $t \notin X$, e non c'è nessuna occorrenza di X in t;
- $X_3 = X_1 \cup X_2$, con $X_1 \not\equiv X_2$ e non ci sono disequazioni della forma $X_i \neq t$ o $t \neq X_i$ in C per ogni $i = 1, 2, 3$;
- $X_1 \parallel X_2$, e $X_1 \neq X_2$;

C è in forma risolta se è vuoto o se tutti i vincoli contenuti in esso sono contemporaneamente in forma risolta.

In JavaSet la risoluzione dei vincoli è affidata alla classe `Solver`. Abbiamo già incontrato alcuni metodi della classe `Solver` negli esempi considerati fino ad

ora. Si è detto che l'introduzione di un singolo vincolo o di una congiunzione di vincoli avviene tramite il metodo statico `add` e che la risoluzione dei vincoli avviene tramite il metodo statico `solve`. La classe `Solver` dispone di altri due metodi per la risoluzione dei vincoli: il metodo `solve1` ed il metodo `setof`. Inoltre dispone di altri metodi di utilità che permettono di stampare le soluzioni ottenute dal metodo `setof` e di visualizzare lo stato dello store.

4.13.1 Risoluzione dei vincoli: il metodo solve

Negli esempi precedenti abbiamo introdotto il metodo `solve`. Questo metodo ricerca, in modo non deterministico, una soluzione che soddisfi tutti i vincoli introdotti. Se i vincoli sono insoddisfacibili, cioè non esiste nessuna soluzione, viene generata un'eccezione `NoSolutionFound` e la computazione termina.

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

Consideriamo il seguente esempio:

Esempio 37

```
class Esempio1 {
 public static void main (String[] args)
 throws IOException,Fallimento {
  Lvar X = new Lvar();     // X = Lvar non inizializzata
  Lvar Y = new Lvar();     // Y = Lvar non inizializzata
  Lvar Z = new Lvar();     // Z = Lvar non inizializzata
  int[] ar = {1,2,3};
  Set s = new Set(ar);                          // s = {1,2,3}
  Set t = new Set(Set.vuoto.ins(Z).ins(Y).ins(X));// t = {X,Y,Z}

  Solver.add(t.eq(s));     // vincolo di uguaglianza
  Solver.solve();
```

```
    System.out.print("\nLvar X = ");Lvar.output(X);
    System.out.print("\nLvar Y = ");Lvar.output(Y);
    System.out.print("\nLvar Z = ");Lvar.output(Z);

    Solver.add(X.neq(1));   // vincolo di disuguaglianza
    Solver.solve();

    System.out.print("\n\nLvar X = ");Lvar.output(X);
    System.out.print("\nLvar Y = ");Lvar.output(Y);
    System.out.print("\nLvar Z = ");Lvar.output(Z);
    return;
   }
  }
```

L'output di questo programma è il seguente:

```
Lvar X = 1
Lvar Y = 2
Lvar Z = 3

Lvar X = 2
Lvar Y = 1
Lvar Z = 3
```

In questo esempio viene introdotto il vincolo di uguaglianza tra gli insiemi `s = {X,Y,Z}` e `t = {1,2,3}` dove `X`, `Y` e `Z` sono variabili logiche non inizializzate. Successivamente viene invocato il metodo `solve`, che trova una possibile soluzione: `X = 1`, `Y = 2` e `Z = 3`. Dopo le istruzioni di stampa, viene aggiunto un nuovo vincolo: `X.neq(1)`, e viene di nuovo invocato il metodo `solve`. Il nuovo vincolo non è soddisfatto dalla soluzione trovata in precedenza, quindi il metodo `solve` applica il backtracking e torna ad un punto di scelta che ha ancora delle alternative aperte e procede nella ricerca di una soluzione, fino a che non ne trova una che soddisfi entrambi i vincoli introdotti. In questo esempio la soluzione trovata dalla seconda invocazione del metodo `solve` è: `X = 2`, `Y = 1` e `Z = 3`. □

Nell'esempio appena considerato, il metodo `solve` è stato invocato due volte, in generale è sufficiente una sola istruzione `Solver.solve()`, dopo che sono stati introdotti tutti i vincoli. A volte però può essere utile o necessario invocare il metodo `solve` più di una volta.

Nel caso in cui il problema considerato richieda l'introduzione di molti vincoli, può accadere che sia necessario molto tempo per trovare un'eventuale inconsistenza, in quanto se i vincoli sono numerosi anche le alternative da esplorare potrebbero essere molte. In questo caso può essere utile aggiungere delle istruzioni `solve`. Ad esempio, supponiamo di dover introdurre i vincoli $v_1, ..., v_p, ..., v_n$, anzichè dare una sola istruzione `Solver.solve()` al termine dell'introduzione di tutti gli n vincoli, potremmo decidere di aggiungere un'invocazione del metodo `solve` subito dopo l'introduzione il p-esimo vincolo. In questo modo se l'inconsistenza è causata dai primi p vincoli, questa viene trovata durante la risoluzione di soli quei vincoli, mentre i vincoli successivi non vengono considerati, e quindi il tempo di esecuzione diminuisce. Se invece i primi p vincoli sono consistenti la computazione continua.

Nel caso di vincoli consistenti i tempi di esecuzione non lievitano con l'aggiunta di istruzioni `solve`, in quanto ogni volta che viene invocato, il metodo `solve` non ricomincia la risoluzione dei vincoli dall'inizio ma riparte dal punto raggiunto dalla `solve` precedente.

Ci sono poi casi in cui l'introduzione di più istruzioni `solve` è necessaria. Si consideri, ad esempio, la classe seguente:

Esempio 38

```
class Esempio2 {
 public static void main (String[] args)
 throws IOException,Fallimento {
  int[] a = {1,2,3};
  Set s = new Set(a);          // S = {1,2,3}
  Lvar R1 = new Lvar();        // R1 = Lvar non inizializzata
  Lvar R2 = new Lvar();        // R2 = Lvar non inizializzata
  Set R = new Set(Set.vuoto.ins(R2).ins(R1)); // R = {R1,R2}
```

```
      Solver.add(R.subset(s));    // vincolo di inclusione
      Solver.solve();

      Solver.add(R1.gt(2));       // vincolo di confronto
      Solver.add(R2.gt(2));       // vincolo di confronto
      Solver.solve();

      System.out.print("\nLvar R1 = ");Lvar.output(R1);
      System.out.print("\nLvar R2 = ");Lvar.output(R2);
      return;
 }
}
```

L'output di questo programma è:

```
  Lvar R1 = 3;
  Lvar R2 = 3;
```

In questo esempio viene introdotto il vincolo `R.subset(s)` dove `s` = {1,2,3} e `R` = {`R1`,`R2`} con `R1` ed `R2` `Lvar` non inizializzate. Successivamente vengono introdotti i vincoli `R1.gt(2)` e `R2.gt(2)`. In questo caso sono necessarie due istruzioni `solve`. Infatti se eliminassimo la prima delle due, al momento della valutazione dei due vincoli `R1.gt(2)` e `R2.gt(2)`, le due variabili logiche `R1` e `R2` non avrebbero ancora nessun valore, quindi verrebbe sollevata un'eccezione `NotIntLvarException` e la computazione terminerebbe. Introducendo invece l'istruzione `Solver.solve()` subito dopo il vincolo di inclusione, al momento della valutazione dei due vincoli di confronto, le due `Lvar` sono già state inizializzate e quindi la valutazione dei vincoli di confronto porta alla soluzione `R1 = 3` e `R2 = 3`. □

In generale, quindi, bisogna introdurre un'istruzione `Solver.solve()` tutte le volte che il risultato della valutazione di alcuni vincoli o più precisamente i valori assegnati ad alcune variabili devono essere noti al momento della valutazione dei vincoli successivi.

4.13.2 Risoluzione dei vincoli: una soluzione

Un altro metodo per la risoluzione dei vincoli è il metodo `solve1`. Questo metodo, come il metodo `solve`, ricerca, in modo non deterministico, una soluzione che soddisfi tutti i vincoli introdotti. Se non c'è nessuna soluzione, viene sollevata un'eccezione `NoSolutionFound` e la computazione termina. Il metodo `solve1`, a differenza del metodo `solve`, non tiene traccia delle alternative rimaste inesplorate, quindi in caso di backtracking, quelle alternative non vengono considerate. Il metodo `solve1`, quindi, si comporta come il *cut* del Prolog, cioè "taglia" delle alternative.

Si consideri il seguente esempio:

Esempio 39

```
class Es1Solve1 {
  public static void main (String[] args)
  throws IOException,Fallimento {
   Lvar X = new Lvar();      // X: Lvar non inizializzata
   Lvar Y = new Lvar();      // Y: Lvar non inizializzata
   Lvar Z = new Lvar();      // Z: Lvar non inizializzata
   int[] ar = {1,2,3};
   Set s = new Set(ar);                          // s = {1,2,3}
   Set t = new Set(Set.vuoto.add(Z).add(Y).add(X));// t = {X,Y,Z}

   Solver.add(t.eq(s));       // vincolo di uguaglianza
   Solver.solve1();

   System.out.print("\nLvar X = ");Lvar.output(X);
   System.out.print("\nLvar Y = ");Lvar.output(Y);
   System.out.print("\nLvar Z = ");Lvar.output(Z);

   Solver.add(X.neq(1));      // vincolo di disuguaglianza
   Solver.solve();

   System.out.print("\nLvar X = ");Lvar.output(X);
   System.out.print("\nLvar Y = ");Lvar.output(Y);
```

```
      System.out.print("\nLvar Z = ");Lvar.output(Z);
      return;
   }
  }
```

In questo esempio viene introdotto il vincolo di uguaglianza tra gli insiemi `s = {X,Y,Z}` e `t = {1,2,3}` dove `X`, `Y` e `Z` sono variabili logiche non inizializzate. Successivamente viene invocato il metodo `solve1`, che trova una possibile soluzione: `X = 1`, `Y = 2` e `Z = 3` e "taglia" le altre alternative. Dopo le istruzioni di stampa, viene aggiunto un nuovo vincolo: `X.neq(1)`, e viene invocato il metodo `solve`. Il nuovo vincolo non è soddisfatto dalla soluzione trovata in precedenza, quindi il metodo `solve` applica il backtracking, ma non trova nessuna alternativa, quindi non trova nessuna soluzione e la computazione si interrompe.

L'output del programma è il seguente:

```
Lvar X = 1
Lvar Y = 2
Lvar Z = 3

    Exception in thread "main" NoSolutionFound
```

□

Consideriamo un altro esempio, in cui viene mostrato un caso in cui l'uso del metodo `solve1` permette di eliminare delle alternative inutili:

Esempio 40

```
class Es2Solve1 {
   public static void main (String[] args)
   throws Fallimento {
     int[] s_val = {1,2,3};
     Lvar X = new Lvar();       // X = Lvar non inizializzata
     Lvar Y = new Lvar();       // Y = Lvar non inizializzata
     Lvar Z = new Lvar();       // Z = Lvar non inizializzata
```

```
    Set s = new Set(s_val);     // s = {1,2,3}
    Set t = new Set(Set.vuoto.add(Z).add(Y).add(X));// t = {X,Y,Z}
    forallIn(t,s);
  }

  public static void forallIn(Set T,Set S)
  {
    if(T.isEmpty())return;
    else{
      Set R = new Set();
      Lvar W = new Lvar();
      less(T,W,R);
      Solver.add(W.in(S));
      Solver.solve1();
      forallIn(R,S);
      return;
    }
  }
}
```

Il metodo `main` della classe `Es2Solve1` definisce i due insiemi `s` = {`1,2,3`} e `t` = {`X,Y,Z`} con `X`, `Y` e `Z` variabili logiche non inizializzate. Il metodo `forallIn` è ricorsivo, la ricorsione termina quando `S` è uguale all'insieme vuoto. Ad ogni passo, l'insieme `S`, su cui viene fatto il controllo, è ottenuto dal metodo `less` del passo precedente, quindi è necessario che ad ogni ricorsione venga eseguita la risoluzione dei vincoli in modo che al passo successivo `S` sia un insieme inizializzato. Il metodo `forallIn` ad ogni ricorsione estrae un elemento dall'insieme {`X,Y,Z`} e aggiunge al constraint store il vincolo di appartenenza di quell'elemento all'insieme {`1,2,3`}.

Si noti che in questo esempio abbiamo definito il metodo `forallIn`, per dare un esempio dell'uso del metodo `solve1`, altrimenti avremmo utilizzato il metodo `forall`.

L'output di questo programma è il seguente:

```
Lvar X = 1
Lvar Y = 2
Lvar Z = 3
```

□

L'uso del metodo `solve1` o del metodo `solve` è indifferente dal punto di vista della soluzione ottenuta, ma il metodo `solve1` permette, in alcuni casi, di eliminare delle alternative inutili. Nell'esempio appena considerato, infatti, il metodo `solve`, nel risolvere la `less`, terrebbe traccia di tutte le possibili alternative per estrarre un elemento dall'insieme. Queste alternative sono inutili, infatti l'ordine con cui vengono estratti gli elementi dall'insieme non è importante, in quanto, nel nostro caso, prima o poi vengono estratti tutti. Quindi con l'uso del metodo `solve1`, in caso di backtracking, si evita di esplorare delle alternative inutili, che non modificherebbero il risultato, ma renderebbero la computazione più costosa.

Il metodo `solve1` deve essere usato solo nei casi in cui si è sicuri di "buttare via" delle alternative inutili. Negli altri casi bisogna utilizzare il metodo `solve` che esegue invece una ricerca completa, senza trascurare nessuna alternativa.

4.13.3 Insieme delle soluzioni

Il metodo statico `setof` della classe `Solver` permette di determinare, per una variabile logica, tutte le possibili soluzioni che soddisfano l'insieme di vincoli dato. Data l istanza della classe `Lvar`, `Lst` o `Set`, $Solver.setof(l)$ restituisce la lista di tutte le possibili soluzioni per l. Per la stampa si utilizza il metodo `output` della classe `Lst`.

Consideriamo il seguente esempio:

Esempio 41

```
class EsSetof {
 public static void main (String[] args) {
   int[] a = {1,2,3,4};
   Set s = new Set(a);          // s = {1,2,3,4}
   Lvar X = new Lvar();         // X = Lvar non inizializzata
```

```
    Set T = new Set();          // T = Set non inizializzato
    Solver.add(X.in(s));        // vincolo di appartenenza
    Solver.add(T.subset(s));    // vincolo di inclusione
    System.out.print(" setof(X) = "); Lst.output(Solver.setof(X));
    System.out.print(" setof(T) = "); Lst.output(Solver.setof(T));
    return;
  }
 }
```

In questo esempio vengono introdotti i vincoli `X.in(s)` e `T.subset(s)` con `X` variabile logica non inizializzata, `T` insieme non inizializzato e `s = {1,2,3,4}`. Le istruzioni `Solver.setof(X)` e `Solver.setof(T)` permettono di determinare tutte le possibili soluzioni per `X` e per `T`. L'insieme delle possibili soluzioni per `X` è dato da tutti gli elementi di `s`, mentre l'insieme delle possibili soluzioni di `T` coincide con l'insieme delle parti di `s`.

Le istruzioni `Lst.output(Solver.setof(X))` e `Lst.output(Solver.setof(T))` permettono di ottenere l'output delle possibili soluzioni per la variabile `X` e per l'insieme `T`.

L'output di questo programma è il seguente:

```
setof(X) = [1,2,3,4]
setof(T) = [{},{4},{3},{3,4},{2},{2,4},{2,3},{2,3,4},{1},{1,4},
           {1,3},{1,3,4},{1,2},{1,2,4},{1,2,3},{1,2,3,4}]
```

□

4.14 Visualizzazione dei vincoli

Il metodo statico `showStore()` della classe `Solver` permette di visualizzare i vincoli contenuti nello store. Questo metodo può essere invocato ovunque nel programma e mostra lo stato dello store nel momento in cui viene invocato. Ad esempio, se nello store sono memorizzati i vincoli $X.eq(2)$ e $s_1.union(s_2, s_3)$ con X istanza della

classe `Lvar` e s_1, s_2, s_3 istanze della classe `Set`, l'output ottenuto con l'istruzione `Solver.showStore()` sarà del tipo: $\widetilde{X}.eq(2)$ e $\tilde{s}_1.union(\tilde{s}_2, \tilde{s}_3)$ dove $\widetilde{X}$, $\tilde{s}_1$, $\tilde{s}_2$ e $\tilde{s}_3$ sono gli identificatori usati dalla JVM per memorizzare X, s_1, s_2 e s_3.

Consideriamo il seguente esempio:

Esempio 42

```
class esShowStore1 {
 public static void main (String[] args)
 throws IOException,Fallimento {
   Lvar X = new Lvar();          // X: Lvar non inizializzata
   Lvar Y = new Lvar();          // Y: Lvar non inizializzata
   Lvar Z = new Lvar();          // Z: Lvar non inizializzata
   int[] ar = {1,2,3};
   Set s = new Set(ar);                          // s = {1,2,3}
   Set t = new Set(Set.vuoto.add(Z).add(Y).add(X));// t = {X,Y,Z}

   Solver.add(t.eq(s));          // vincolo di uguaglianza
   Solver.add(X.neq(1));         // vincolo di disuguaglianza

   System.out.println("Stato dello store prima della risoluzione: ");
   Solver.showStore();

   Solver.solve();
   System.out.println("Soluzione: ");
   System.out.print("\nLvar X = ");Lvar.output(X);
   System.out.print("\nLvar Y = ");Lvar.output(Y);
   System.out.print("\nLvar Z = ");Lvar.output(Z);

   System.out.println("Stato dello store dopo la risoluzione: ");
   Solver.showStore();
   return;
 }
}
```

In questo esempio vengono introdotti il vincolo di uguaglianza `t.eq(s)` e il vincolo di disuguaglianza `X.neq(1)` con `t = {X,Y,Z}`, `s = {1,2,3}` e `X`, `Y` e `Z` variabili logiche non inizializzate. Quindi, in termini più astratti i vincoli sono: $\{X, Y, Z\} = \{1, 2, 3\} \wedge X \neq 1$.

Nel programma ci sono due istruzioni `Solver.showStore()`: una prima della istruzione `Solver.solve()` e una dopo.

L'output del programma è il seguente:

```
Stato dello store prima della risoluzione:
  Set@20c10f.eq(Set@62eec8)
  Lvar@2a9835.neq(1)

Soluzione:
  Lvar X = 2
  Lvar Y = 1
  Lvar Z = 3

Stato dello store dopo la risoluzione:
  1.eq(1)
  Set@136228.eq(Set@136228)
  2.eq(2)
  3.eq(3)
  Set@113750.eq(Set@113750)
  Lvar@2a9835.neq(1)
```

Quando il metodo `showStore` viene invocato per la prima volta, nello store ci sono solo i due vincoli introdotti: `t.eq(s))` e `X.neq(1)`, che devono ancora essere risolti, quindi con la prima invocazione del metodo `showStore` si ottiene la stampa di questi due vincoli: `Set@20c10f.eq(Set@62eec8)` e `Lvar@2a9835.neq(1)`, dove `Set@20c10f` è l'identificatore dell'insieme $\{X, Y, Z\}$, `Set@62eec8` è l'identificatore dell'insieme $\{1, 2, 3\}$ e `Lvar@2a9835` è l'identificatore di X. La seconda invocazione avviene dopo la risoluzione, cioè dopo che i vincoli sono stati trasformati, dal risolu-

tore, in forma risolta. Quindi la seconda invocazione di `showStore` mostra lo stato dello store al termine della risoluzione dei vincoli. □

Consideriamo un altro esempio:

Esempio 43

```
class esShowStore2 {
 public static void main (String[] args)
 throws IOException,Fallimento{
   int[] a = {1,2};
   Set s = new Set(a);              // s = {1,2}
   Set R = new Set();               // R = Set non inizializzato
   Solver.add(R.subset(s));         // vincolo di inclusione
   Solver.add(R.neq(Set.vuoto));    // vincolo di disuguaglianza

   System.out.println("Stato dello store prima della risoluzione: ");
   Solver.showStore();

   Solver.solve();
   System.out.println("Soluzione: ");
   System.out.print("\nSet R = ");Set.output(R);
   System.out.println("Stato dello store dopo la risoluzione: ");
   Solver.showStore();
   return;
 }
}
```

In questo esempio vengono introdotti il vincolo di inclusione `R.subset(s)` e il vincolo di disuguaglianza `R.neq(Set.vuoto)` con `s = {1,2}`, `R` insieme non inizializzato. Anche in questo caso abbiamo introdotto due istruzioni per la stampa dello store, una prima di risolvere i vincoli e una dopo.

L'output del programma è il seguente:

```
Stato dello store prima della risoluzione:
  Set@62eec8.union(Set@20c10f,Set@62eec8)
  Set@20c10f.neq(Set@2a9835)

Soluzione:
  Set R = {2}

Stato dello store dopo la risoluzione:
  1.eq(1)
  Set@4672d0.eq(Set@4672d0)
  1.nin(Set@2a9835)
  1.eq(1)
  Set@1bd03e.eq(Set@1bd03e)
  Set@4abc9.neq(Set@2a9835)
  1.nin(Set@2a9835)
  2.eq( 2)
  Set@2a9835.eq(Set@2a9835)
  2.nin(Set@2a9835)
  Set@4abc9.eq(Set@4abc9)
  2 .eq(2)
  Set@2a9835.eq(Set@2a9835)
  1.neq(2)
  1.neq(2)
  2.nin(Set@2a9835)
  2.nin(Set@2a9835)
  Set@2a9835.eq(Set@2a9835)
  Set@2a9835.eq(Set@2a9835)
```

Quando il metodo `showStore` viene invocato per la prima volta, nello store ci sono solo i due vincoli: `R.subset(s))` e `R.neq(Set.vuoto)`, che devono ancora essere risolti. Il metodo `showStore` stampa i due vincoli: `Set@62eec8.union(Set@20c10f,Set@62eec8)` e `Set@20c10f.neq(Set@2a9835)`. Il primo è un vincolo di unione, in quanto abbiamo definito il vincolo `subset` tramite il vincolo `union`, quindi `R.subset(s)` viene memorizzato come `s.union(R,s)`.

La seconda invocazione del metodo `showStore` avviene dopo la risoluzione, cioè

dopo che i vincoli sono stati trasformati, dal risolutore, in forma risolta. Come si può notare, la risoluzione di un vincolo di unione comporta l'aggiunta di numerosi vincoli nello store. □

La stampa dello store, così come è realizzata ora, è poco utile perchè spesso è difficile capire a quale variabile si riferisce ciascun identificatore. Questo è un punto che potrà essere migliorato in futuro. Un'idea potrebbe essere quella di assegnare un nome a ciascuna variabile, ad esempio aggiungendo un campo di tipo `String` alle classi `Lvar`, `Lst` e `Set` e passando al costruttore il nome da assegnare alla variabile che si sta creando. Ad esempio, una `Lvar X` non inizializzata e una `Lvar Y` inizializzata con `1` potrebbero essere create come segue:

```
Lvar X = new Lvar("X");
Lvar Y = new Lvar(1,"Y");
```

Bisognerebbe però assegnare un nome, non solo alle variabili create dall'utente, ma anche a quelle che vengono create dal risolutore durante la risoluzione dei vincoli. Il risolutore potrebbe generare un nome diverso per ciascuna nuova variabile che crea. La `showStore` potrebbe poi stampare il nome assegnato all'oggetto anziché stampare l'identificatore usato dalla macchina virtuale per identificarlo.

Capitolo 5

Implementazione di JavaSet

5.1 Introduzione

`JavaSet` è una libreria realizzata come un package Java. In questo capitolo ci occuperemo dell'implementazione delle classi più importanti della libreria e di come, attraverso le classi di JavaSet, abbiamo cercato di riprodurre alcuni costrutti, tipici nei linguaggi CLP, che non sono presenti in Java: le variabili logiche, la risoluzione dei vincoli, il non determinismo.

Le classi principali di JavaSet sono:

- **Lvar** per le variabili logiche.
- **Lst** per le liste.
- **Set** per gli insiemi.
- **Solver** per la risoluzione dei vincoli.

Queste classi sono dichiarate `public`, cioè chiunque può dichiarare dei riferimenti a oggetti di queste classi o accedere ai loro membri pubblici.

Appartengono alla libreria anche le seguenti classi:

- **StoreElem** per la memorizzazione e la gestione di vincoli.
- **VetStoreElem** per i vettori di vincoli (questa classe estende la classe Vector).
- **ChoicePoint** per i punti di scelta.
- **StatoStore** per memorizzare lo stato dello store nei punti di scelta.

- **StatoVar** per memorizzare lo stato delle variabili nei punti di scelta.

Queste classi non hanno nessun modificatore di accesso, quindi sono accessibili esclusivamente dalle classi di JavaSet.

Infine elenchiamo le classi di tipo eccezione:

- **EmptyLstException**
- **Fallimento**
- **InizLvarException**
- **NoSolutionFound**
- **NotInizVarException**
- **NotIntLvarException**
- **NotLstException**
- **NotSetException**
- **VarNonIniz**

La classe `Fallimento`, in particolare, estende la classe `Exception`, cioè crea eccezioni controllate, quindi deve essere dichiarata nella clausola `throws` dei metodi che sollevano eccezioni di questo tipo. Tutte le altre estendono la classe `RuntimeException`; in questo modo creano eccezioni non controllate, ovvero che possono essere sollevate senza dover essere dichiarate nelle clausole `throws`.

In `JavaSet` le variabili logiche, le liste e gli insiemi sono oggetti, rispettivamente, delle classi `Lvar`, `Lst` e `Set`. Questi oggetti hanno diversi campi nascosti, tra i quali un campo che indica se la variabile è inizializzata o no e uno in cui è memorizzato il suo valore, nel caso in cui ne abbia uno. I vincoli sono oggetti della classe `StoreElem` e i loro attributi contengono le informazioni relative agli elementi coinvolti nel vincolo e al tipo di vincolo definito su di essi. Ogni volta che viene definito un vincolo, questo viene memorizzato nello *store*, che è un vettore della classe `Solver`.

La risoluzione dei vincoli è affidata alla classe `Solver`. Gli algoritmi di risoluzione che abbiamo adottato sono quelli di $SAT_{s\varepsilon\tau}$ cioè del risolutore di vincoli del linguaggio CLP($\mathcal{SET}$) [8].

La risoluzione avviene in modo non deterministico. Il risolutore cerca di ridurre lo store ad una forma risolta, per poter determinare se è soddisfacibile. Tutte le volte che per un vincolo si presentano diverse possibili soluzioni, il risolutore procede creando un punto di scelta.

Un punto di scelta è un oggetto della classe `ChoicePoint` provvisto di diversi campi nei quali vengono memorizzati il vincolo su cui ci sono delle alternative aperte, e lo stato delle variabili e dello store in quel momento. I punti di scelta vengono memorizzati in una pila, cioè in una struttura dati di tipo LIFO. A questo punto il risolutore procede considerando una delle possibili soluzioni per quel vincolo. Se, in seguito, la soluzione considerata causa un'inconsistenza, l'algoritmo applica il backtracking, cioè estrae un punto di scelta dalla pila, riporta le variabili e lo store nello stato memorizzato in quel punto di scelta e continua esaminando un'altra alternativa. Se al momento dell'estrazione del punto di scelta la pila è vuota, significa che non ci sono più alternative possibili, quindi viene sollevata un'eccezione di tipo `NoSolutionFound` e la computazione termina.

5.2 Le variabili logiche

In `JavaSet` una variabile logica è un oggetto della classe `Lvar`. Questa classe è definita `public`, mentre i suoi attributi sono dichiarati `protected`. In questo modo chiunque può dichiarare dei riferimenti a oggetti di tipo `Lvar` o accedere ai membri `public`, ma gli attributi di questi oggetti sono accessibili solo dai metodi delle classi appartenenti al package `JavaSet`.

I metodi delle classi `Lvar`, sono in parte `public`, cioè accessibili da tutte le classi, e in parte `protected`, cioè accessibili solo dalle classi del package.

Tra i metodi della classe `Lvar` sono definiti tutti i metodi non statici della classe `Lst` e della classe `Set`, in quanto, nel caso in cui il valore di una `Lvar` sia una lista o un insieme, deve essere possibile eseguire su questa `Lvar` le stesse operazioni eseguibili su un oggetto `Lst` o `Set`.

Si potrebbe pensare che la classe `Lvar` sia stata realizzata come estensione delle

classi `Lst` e `Set`. Non è così, Java non supporta l'ereditarietà multipla e quindi la classe `Lvar` non può essere una sottoclasse sia della classe `Lst` che della classe `Set`. Tra le classi della libreria non è definita nessuna relazione di ereditarietà.

Ogni variabile logica ha diversi attributi nascosti che contengono i dati associati a quella variabile e ne definiscono lo stato durante l'esecuzione di un programma.

Di seguito riportiamo gli attributi della classe `Lvar`:

```
protected boolean iniz = false;
protected Object val;
protected Lvar equ = null;
protected static Vector nonInizLvar = new Vector();
```

L'attributo `iniz` indica se la variabile è inizializzata oppure no. Sia `X` un'istanza della classe `Lvar`, se `X` non è inizializzata, allora `X.iniz` è uguale a `false`, mentre se è inizializzata è uguale a `true`. Per default il campo `iniz` viene inizializzato con il valore `false`.

Il campo `val` contiene il valore della `Lvar`, nel caso in cui sia inizializzata. Se la variabile non è inizializzata, il campo è uguale a `null`.

Si noti che `val` è una variabile di tipo `Object` e, poiché la classe `Object` è alla radice della gerarchia delle classi, l'attributo `val` può riferirsi a qualunque oggetto venga passato come parametro al costruttore della `Lvar`, in particolare anche a istanze della classe `Lst` e `Set`. Tuttavia `val` non può contenere tipi primitivi, perchè non rientrano in nessuna classe. Nel caso in cui il valore di una `Lvar` sia di tipo primitivo si ricorre alle classi "avvolgenti", cioè l'attributo `val` di una `Lvar` inizializzata con un tipo primitivo, è un riferimento ad un oggetto della corrispondente classe wrapper.

Il campo `equ` è un riferimento ad una variabile logica. Questo attributo è stato introdotto per rendere più efficiente la propagazione dei vincoli. Infatti si supponga di aver definito n variabili logiche, ad esempio, istanze della classe `Lvar`: L_1, L_2,...,L_n. Supponiamo che le n `Lvar` siano tutte variabili non inizializzate e che siano vincolate ad essere tutte uguali. Non è conveniente considerare queste variabili come

n variabili distinte, è molto più naturale considerarle come riferimenti ad un'unica variabile, in quanto qualunque operazione eseguita su una di queste `Lvar` deve avere effetto anche su tutte le altre; questo è reso possibile dall'attributo `equ`. Ad esempio, supponiamo che sia stato definito il vincolo $L_i.eq(3)$ dove i è un qualsiasi intero tra 1 e n, questo vincolo dove essere propagato a tutte le n `Lvar`, in quanto sono legate tra loro dalla relazione di uguaglianza. L'attributo `equ` ci permette di propagare il vincolo senza dover assegnare il valore 3 al campo `val` di tutte le n `Lvar`.

Il campo `equ` di una variabile logica è `null` per default e una volta che gli è stato assegnato un valore diverso da `null`, non può più essere modificato, se non durante il backtracking. L'assegnamento avviene durante la risoluzione dei vincoli di uguaglianza: siano X e Y due `Lvar` con $X.equ$ ed $Y.equ$ entrambi uguali a `null` e sia definito il vincolo $X.eq(Y)$; questo vincolo, se è soddisfacibile, modifica il campo `equ` di X assegnandogli Y. Quindi quando il campo `equ` di una variabile è un riferimento ad un'altra variabile significa che le due variabili sono vincolate ad essere uguali.

Tutte le volte che viene invocato un metodo su una `Lvar` (o una `Lvar` viene passata come parametro ad un metodo), questo metodo verifica se il campo `equ` di quella variabile è uguale a `null`: se lo è procede nell'esecuzione del metodo, in caso contrario applica il metodo alla variabile a cui si riferisce il campo `equ`.

Ad esempio, siano X, Y e Z tre `Lvar` con $X.equ$ uguale a `null`, $Y.equ$ uguale a X e $Z.equ$ uguale a Y. Qualunque metodo, applicato a Z, non agisce direttamente su Z, in quanto il suo campo `equ` è diverso da `null`. Poiché $Z.equ$ è uguale a Y, il metodo viene applicato a Y, ma nemmeno $Y.equ$ è `null`, quindi il metodo viene applicato a X. Quindi Y e Z vengono trattate come riferimenti a X.

Consideriamo ora il seguente esempio. Siano V, W, X, Y, Z, cinque `Lvar`. Supponiamo ora di introdurre i seguenti vincoli:

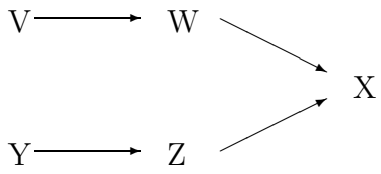
$$V.eq(W)$$

$$W.eq(X)$$

$$Y.eq(Z)$$

$$Z.eq(X)$$

Il primo vincolo assegna ad $V.equ$ la variabile W, il secondo assegna al campo $W.equ$ la variabile X, il terzo assegna a $Y.equ$ la variabile Z e in fine il quarto vincolo assegna a $Z.equ$ la variabile X. Se per indicare che $V.equ$ è uguale a W usiamo la notazione $V \rightarrow W$ allora lo stato dei campi equ delle variabili può essere schematizzato nel modo seguente:

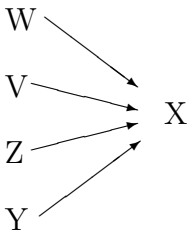


Se introduciamo gli stessi vincoli in un altro ordine, può succedere che cambiano i valori dei campi `equ` delle variabili, pur non cambiando il fatto che al termine della risoluzione dei vincoli tutte le variabili sono uguali tra loro.

Ad esempio supponiamo di introdurre gli stessi vincoli nell'ordine seguente:

$$W.eq(X)$$
$$V.eq(W)$$
$$Z.eq(X)$$
$$Y.eq(Z)$$

Si ha che il primo vincolo assegna a $W.equ$ la variabile X, il secondo vincolo assegna al campo $Z.equ$ non W ma X in quanto $W.equ$ e un riferimento a X. La stessa cosa avviene per Z e Y. Quindi al termine della risoluzione di questi vincoli la situazione può essere schematizzata come segue.



Si noti che, con il campo `equ`, abbiamo riprodotto gli effetti di un assegnamento, cioè assegnare a $X.equ$ la variabile Y produce lo stesso effetto dell'assegnamento

$X = Y$, ma mentre il valore del campo `equ` può essere modificato in fase di backtracking, un'assegnamento, invece, non potrebbe essere annullato.

Infine la classe `Lvar` possiede un attributo statico: il vettore `NonInizLvar`. Gli elementi di questo vettore sono riferimenti a variabili logiche non inizializzate o che erano tali al momento della loro creazione. Come vedremo, questo vettore viene utilizzato tutte le volte che viene creato un punto di scelta.

5.3 Le liste e gli insiemi

In `JavaSet` una lista è un oggetto della classe `Lst`. Questa classe è definita `public`, mentre i suoi attributi sono dichiarati `protected`. In questo modo chiunque può dichiarare dei riferimenti a oggetti di tipo `Lst` o accedere ai membri `public` della classe, ma gli attributi di questi oggetti sono accessibili solo dai metodi delle classi appartenenti al package `JavaSet`.

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

Ogni oggetto `Lst` ha diversi attributi nascosti che contengono i dati associati a quella lista e ne definiscono lo stato durante l'esecuzione di un programma.

Di seguito riportiamo gli attributi della classe `Lst`:

```
protected boolean iniz = false;
protected Vector lista;
protected Lst resto;
protected Lst equ = null;
protected static final Lst vuoto = new Lst(null,null);
protected static Vector nonInizLst = new Vector();
```

Gli attributi `iniz` ed `equ` sono analoghi agli stessi campi della classe `Lvar`.

I campi `lista` e `resto` servono per memorizzare il valore della lista. L'attributo `lista` è un riferimento ad un vettore, mentre l'attributo `resto` è un riferimento ad un oggetto `Lst`.

Ad esempio, consideriamo le liste `R`, `s`, `t` e `u` definiti come segue:

Esempio 44

```
int[] ar = {3,4};                  // array di interi
Lst R = new Lst();                 // R: lista non inizializzata
Lst s = new Lst(R.ins(2).ins(1)); // s = [1,2|R]
Lst t = new Lst(R.insAll(ar));     // t = [3,4|R]
Lst u = new Lst(Lst.vuoto.insAll(ar)); // u = [3,4]
```

`R.lista` e `R.resto` sono entrambi `null` in quanto `R` è una lista non inizializzata. Il campo `s.lista` è un riferimento ad un vettore che contiene solo l'intero `1`, `s.resto` è un riferimento alla lista `[2|R]`. A sua volta il campo `lista` della lista `[2|R]` è un riferimento al vettore con il solo elemento `2`, mentre il campo `resto` è un riferimento alla lista `R`. L'attributo `t.lista` è un riferimento al vettore contenente `3` e `4`, mentre `t.resto` è un riferimento alla lista `R`. Infine il campo `lista` dell'insieme `u` è dato dal vettore contenente `3` e `4` e il suo `resto` è la lista vuota. □

Quindi una lista `s` è limitata se `s.resto` è la lista vuota o una lista limitata, mentre è non limitata se `s.resto` è una lista non inizializzata o non limitata.

La classe `Lst` possiede, poi, l'attributo `vuoto`. Questo attributo è dichiarato `static`, cioè ne esiste un'unica copia indipendentemente dal numero di oggetti `Lst` che vengono creati, inoltre `vuoto` è dichiarato `final`, cioè non può essere modificato. L'attributo `vuoto` definisce la lista vuota. Questo attributo è un riferimento ad una lista i cui campi `lista` e `resto` sono uguali a `null`, come per una lista non inizializzata, ma il cui campo `iniz` è `true`.

Infine la classe `Lst` possiede un attributo statico dato dal vettore `nonInizLst`. Gli elementi di questo vettore sono riferimenti a liste non inizializzate o che erano tali al momento della loro creazione. Questo vettore viene utilizzato quando viene creato un punto di scelta.

In `JavaSet` un insieme è un oggetto della classe `Set`. Come `Lst`, anche la classe `Set` è definita `public`, mentre i suoi attributi sono dichiarati `protected`.

Ogni oggetto `Set` ha diversi attributi nascosti che contengono i dati associati a quell'insieme e ne definiscono lo stato durante l'esecuzione di un programma.

Di seguito riportiamo gli attributi della classe `Set`:

```
protected boolean iniz = false;
protected Vector elem;
protected Set resto;
protected Set equ = null;
protected static final Set vuoto = new Set(null,null);
protected static Vector nonInizSet = new Vector();
```

Gli attributi `iniz` ed `equ` sono analoghi a quelli delle classi `Lvar` e `Lst`.

I campi `elem` e `resto` sono analoghi agli attributi `lista` e `resto` della classe `Lst` e servono per memorizzare il valore dell'insieme.

Ad esempio, consideriamo gli insiemi `R`, `s`, `t` e `u` definiti come segue:

Esempio 45

```
int[] ar = {3,4};                  // array di interi
Set R = new Set();                 // R: insieme non inizializzato
Set s = new Set(R.ins(2).ins(1)); // s = {1,2|R}
Set t = new Set(R.insAll(ar));     // t = {3,4|R}
Set u = new Set(Set.vuoto.insAll(ar)); // u = {3,4}
```

`R.elem` e `R.resto` sono entrambi `null` in quanto `R` è un insieme non inizializzato. Il campo `s.elem` è un riferimento ad un vettore che contiene solo l'intero `1`, `s.resto` è un riferimento all'insieme {`2|R`}. A sua volta il campo `elem` dell'insieme {`2|R`} è un riferimento al vettore con il solo elemento `2`, mentre il campo `resto` è un riferimento alla lista `R`. L'attributo `t.elem` è un riferimento al vettore contenente `3` e `4`, mentre `t.resto` è un riferimento all'insieme `R`. Infine il campo `elem` dell'insieme `u` è dato dal vettore contenente `3` e `4` e il suo `resto` è l'insieme vuoto. □

Quindi un insieme `s` è limitato se `s.resto` è l'insieme vuoto o un insieme limitato, mentre è non limitato se `s.resto` è un insieme non inizializzato o non limitato.

Anche la classe `Set` possiede l'attributo `vuoto`. che definisce l'insieme vuoto.

Infine la classe `Set` possiede un attributo statico dato dal vettore `nonInizSet` con lo stesso uso del vettore `nonInizLst` della classe `Lst`.

5.4 Inserimento ed estrazione da liste e insiemi

Nelle classi `Lvar`, `Lst` e `Set` sono previsti diversi metodi public che permettono di aggiungere uno o più elementi ad un insieme dato o ad una lista data e di estrarre un elemento dalla testa o dalla coda di una lista. Nessuno di questi metodi modifica l'insieme o la lista sulla quale sono invocati, tutti costruiscono e restituiscono un nuovo insieme o una nuova lista.

Come tutti gli altri metodi pubblici e non statici delle classi `Lst` e `Lvar`, i metodi di inserimento ed estrazione delle classi `Set` e `Lst` sono implementati anche nella classe `Lvar`, in quanto deve essere possibile inserire o estrarre elementi anche da una `Lvar` se il suo valore è un insieme o una lista.

E' importante osservare che avremmo potuto definire tutti i metodi e i vincoli definiti su liste e insiemi come metodi e vincoli soltanto della classe `Lvar` e permettere così l'utilizzo di insiemi e liste solo se contenuti in una `Lvar`. In questo modo le variabili logiche sarebbero state gli unici oggetti su cui operare e su cui definire i vincoli. Tuttavia l'utilizzo di liste e insiemi sarebbe stato più difficoltoso e quindi abbiamo permesso l'utilizzo di insiemi e liste anche senza l'obbligo di "avvolgerli" in una `Lvar`.

5.4.1 Inserimento di elementi in una lista

I metodi di inserimento della classe `Lst` sono i seguenti:

```
public Lst ins1(type elem)
public Lst ins1All(type[] ar)
public Lst insn(type elem)
public Lst insnAll(type[] ar)
```

In Java è possibile effettuare l'overloading dei metodi, quindi abbiamo potuto sovraccaricare questi metodi con diverse implementazioni che si differenziano solo per il tipo del parametro passato. In seguito, con *type* indicheremo i seguenti tipi: `char`, `boolean`, `int`, `byte`, `short`, `long`, `float`, `double`, `Character`, `Boolean`, `Number`, `Lvar`, `Lst`, `Set`.

Riportiamo, di seguito, l'implementazione del metodo `ins1` della classe `Lst` nel caso in cui il parametro passato sia un `int`, e l'implementazione del metodo `insn` nel caso in cui il parametro sia un `Set`. Negli altri casi l'implementazione è del tutto simile.

```
public Lst ins1(int i)
{
 if(equ == null){
    Vector v = new Vector();
    Integer ii = new Integer(i);
    v.add(ii);
    Lst s = new Lst(v,this);
    return s;
 }
 else return this.equ.ins1(i);
}

public Lst insn(Set s)
{
 if(this.equ == null){
    Vector vet = this.toVector();
    for(int i = 0; i < arr.length; i++){
        vet.add(arr[i]);
    }
    Lst lst = new Lst(vet,vuoto);
    return lst;
 }
 else return this.equ.insnAll(arr);
}
```

Il metodo `ins1` definisce un vettore vuoto `v` al quale viene aggiunto il parametro

passato (oppure tutti gli elementi dell'array passato, nel caso del metodo `ins1All`). Se il parametro è di tipo primitivo, viene prima trasformato in un oggetto della corrispondente classe avvolgente, in questo modo tutti gli elementi di un insieme sono oggetti.

Il metodo crea e restituisce una nuova `Lst s` tale che `s.lista` è un riferimento a `v` e `s.resto` è un riferimento a `this`, cioè alla lista su cui è stato invocato il metodo.

Il metodo `insn` definisce un vettore `vet` ottenuto come risultato del metodo `toVector` applicato alla lista corrente. Il metodo `toVector` è un metodo `protected`. Questo metodo restituisce il vettore contenente tutti gli elementi della lista o dell'insieme su cui è invocato. A questo vettore viene poi aggiunto, in coda, il parametro passato al metodo `insn` (oppure tutti gli elementi dell'array passato al metodo `insnAll`). Se il parametro è di tipo primitivo viene prima trasformato in un oggetto della corrispondente classe avvolgente. Il metodo crea e restituisce una nuova `Lst` tale che il suo campo `lista` è un riferimento a `vet` e il campo `resto` è un riferimento alla lista su cui è stato invocato il metodo.

Consideriamo ora l'implementazione del metodo `ins1` della classe `Lvar`, nel caso in cui il parametro sia un `byte`. Per gli altri tipi e per gli altri metodi l'implementazione è del tutto simile:

```
public Lst ins1(byte i)
{
 if(equ == null){
    if(this.val instanceof Lst)return ((Lst)this.val).ins1(i);
    else throw new NotLstException();
 }
 else return this.equ.ins1(i);
}
```

I metodi `ins1` e `ins1All` `insn` e `insnAll`, se applicati ad una `Lvar`, verificano se il valore della variabile è un'istanza della classe `Lst` oppure no. Se lo è, i metodi restituiscono il risultato ottenuto applicando i metodi a questa lista, se invece il

valore della `Lvar` non è una lista, viene sollevata un'eccezione `NotLstException` e la computazione si interrompe .

5.4.2 Inserimento di elementi in un insieme

I metodi di inserimento della classe `Set` sono i seguenti :

```
public Set ins(type elem)
public Set insAll(type [] ar)
```

L'implementazione dei metodi `ins` e `insAll` è del tutto simile all'implementazione dei metodi `ins1` e `ins1All`, ma in questo caso viene creato e restituito un insieme anzichè una lista.

Riportiamo ad esempio l'implementazione del metodo `ins`, della classe `Set`, nel caso in cui il parametro passato sia un intero:

```
public Set ins(int i)
{
 if(equ == null){
    Vector v = new Vector();
    Integer ii = new Integer(i);
    v.add(ii);
    Set s = new Set(v,this);
    return s;
 }
 else return this.equ.ins(i);
}
```

Questo metodo definisce un vettore vuoto `v` al quale viene aggiunto il parametro passato. Se il parametro è di tipo primitivo, viene prima trasformato in un oggetto della corrispondente classe avvolgente, in questo modo tutti gli elementi di un insieme sono oggetti.

Il metodo crea e restituisce un nuovo `Set s` tale che `s.lista` è un riferimento a `v` e `s.resto` è un riferimento `this`, cioè all'insieme su cui è stato invocato il metodo.

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

Consideriamo ora l'implementazione del metodo `ins`, della classe `Lvar`, nel caso in cui il parametro passato sia un'istanza dalla classe `Number` (cioè un oggetto di tipo `Byte`, `Short`, `Integer`, `Long`, `Float` o `Double`). Negli altri casi l'implementazione è del tutto simile:

```
public Set ins(Number n)
{
 if(equ == null){
  if(this.val instanceof Set)return ((Set)this.val).ins(n);
  else throw new NotSetException();
 }
else return this.equ.ins(n);
}
```

I metodi `ins` ed `insAll`, se applicati ad una `Lvar`, verificano se il valore della variabile è un'istanza della classe `Set` e in questo caso applicano a questa istanza il metodo `ins` o `insAll`, altrimenti solleva un'eccezione `NotSetException` che interrompe la computazione.

5.4.3 Estrazione di elementi da una lista

I metodi per l'estrazione di un elemento da una lista sono i seguenti:

```
public Lst ext1(Lvar l)
public Lst extn(Lvar l)
```

L'implementazione dei metodi `ext1` e `extn` della classe `Lst`, è la seguente:

```
public Lst ext1(Lvar l)
{
 if(this.equ == null){
  if(!l.iniz){
    Vector vet = this.toVector();
```

```
    if(vet.get(0) instanceof Lvar)l = (Lvar)vet.get(0);
    else{
       l.iniz = true;
       l.val = vet.get(0);
    }
    vet.removeElementAt(0);
    Lst lst = new Lst(vet,this.resto);
    return lst;
  }
  else throw new InizLvarException();
 }
 else return this.equ.ext1(l);
}


public Lst extn(Lvar l)
{
 if(this.equ == null){
  if(!l.iniz){
    Vector vet = this.toVector();
    if(vet.get(vet.size() - 1) instanceof Lvar)l = (Lvar)vet.get(0);
    else{
       l.iniz = true;
       l.val = vet.get(vet.size()-1);
    }
    vet.removeElementAt(vet.size()-1);
    Lst lst = new Lst(vet,this.resto);
    return lst;
  }
  else throw new InizLvarException();
 }
 else return this.equ.extn(l);
}
```

Questi metodi definiscono un vettore `vet` ottenuto come risultato del metodo `toVector` applicato alla lista corrente. Da questo vettore viene poi estratto il primo o l'ultimo elemento, che viene assegnato alla `Lvar` non inizializzata passata come parametro ai metodi. Se la `Lvar` passata come parametro è inizializzata, viene

generata un'eccezione `InizLvarException` e la computazione termina.

L'implementazione di questi metodi nella classe `Lvar` è del tutto simile all'implementazione vista in precedenza per gli altri metodi.

Si noti che, poiché Java non supporta l'overloading degli operatori, in JavaSet non è stato possibile realizzare le operazioni di inserimento ed estrazione in modo più naturale, utilizzando cioè degli operatori al posto dei metodi alfanumerici, come avviene invece in SINGLETON [16].

Ad esempio, in SINGLETON, per inserire in testa alla lista l l'elemento e si usa l'espressione $e +> l$, e per inserire l'elemento in coda alla lista si usa l'espressione $l <+ e$. In JavaSet invece queste espressioni corrispondono rispettivamente a: $new\ Lst(l.ins1(e))$ e $new\ Lst(l.insn(e))$. Per estrarre l'elemento e dalla testa o dalla coda della lista l, in SINGLETON, si usano le espressioni $e <\!\!- l$, e $l -\!\!> e$, che in JavaSet corrispondono a: $new\ Lst(l.ext1(e))$ e $new\ Lst(l.extn(e))$.

Infine in SINGLETON per aggiungere gli elementi e_1, e_2 ed e_3 all'insieme s si usa l'espressione $e_1 >> e_2 >> e_3 >> s$, invece in JavaSet si deve usare: $new\ Set(s.ins(e_1).ins(e_2).ins(e_3))$.

5.5 Lo "store"

In JavaSet un singolo vincolo è un oggetto della classe `StoreElem`, mentre una congiunzione di vincoli è un oggetto di tipo `VetStoreElem`, cioè un vettore i cui elementi sono i vincoli della congiunzione.

Di seguito riportiamo gli attributi della classe `StoreElem`:

```
Object obj1;
Object obj2;
Object obj3;
int cons;
int caseControl = 0;
```

Ogni istanza della classe ha tre campi di tipo `Object` in cui vengono memorizzati

gli elementi coinvolti nel vincolo. Nel caso di vincoli binari vengono usati solo due di questi attributi. La classe possiede poi altri due campi: l'attributo intero `cons` in cui viene memorizzato il tipo di vincolo, e l'attributo `caseControl` che per default è uguale a `0`. Quest'ultimo, come vedremo, viene utilizzato nel caso in cui la risoluzione del vincolo richieda una scelta non deterministica.

La classe `VetStoreElem` estende la classe `Vector`. Utilizzando il metodo `and` di questa classe è possibile definire vincoli ottenuti come congiunzione di altri vincoli.

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

```
public VetStoreElem and(StoreElem s)
{
 super.add(s);  // metodo add della classe Vector
 return this;
}
```

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

Gli attributi della classe `Solver` sono i seguenti:

```
private static Vector store = new Vector();
private static Stack alternative = new Stack();
private static boolean storeInvariato = true;
private static int storeSize = 0;
```

Questi attributi sono accessibili solo dalla classe `Solver` stessa e sono tutti statici, cioè definiscono delle variabili globali.

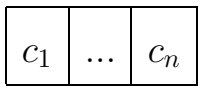
Il constraint store è realizzato mediante l'attributo `store`. Questo attributo è un oggetto di tipo `Vector`, i cui elementi sono riferimenti a istanze della classe `StoreElem`.

L'inserimento di un singolo vincolo, o di una congiunzione di vincoli, nello `store` è realizzato dal metodo statico `add` della classe `Solver`, che riportiamo di seguito:

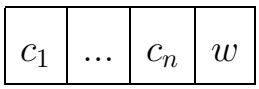
```
public static void add(StoreElem s)
{
 store.add(s);    // metodo add della classe Vector
 return;
}

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

Il metodo `add` è stato sovraccaricato: il parametro passato alla `add` può essere un oggetto di tipo `StoreElem`, cioè un singolo vincolo, oppure un'oggetto di tipo `VetStoreElem`, cioè una congiunzione di vincoli. Nel primo caso il metodo aggiunge in coda al vettore `store` il vincolo passato come parametro, nel secondo caso aggiunge in coda allo `store` tutti i vincoli della congiunzione.

Consideriamo ad esempio il caso in cui nello store siano stati introdotti i vincoli $c_1, ..., c_n$. Lo store può essere schematizzato come segue.



Se poi viene introdotto il vincolo w mediante l'istruzione $Solver.add(w)$ allora lo store diventa:



Se infine viene aggiunto allo store il vincolo dato dalla congiunzione dei vincoli $v_1, ..., v_p$, mediante l'istruzione $Solver.add((v_1).and(v_2)...and(v_p))$, allora lo store diventa:

c_1	...	c_n	w	v_1	v_2	...	v_p

5.6 Risoluzione dei vincoli

La risoluzione dei vincoli introdotti nello store avviene mediante il metodo `solve` della classe `Solver`. Questo metodo cerca di trasformare ciascun vincolo in forma risolta. Se questo processo ha successo significa che l'insieme dei vincoli introdotti ha almeno una soluzione.

In molti casi la riscrittura dei vincoli in forma risolta richiede una scelta non deterministica tra diverse alternative. Durante la risoluzione, il metodo `solve` tiene traccia delle alternative rimaste inesplorate. In questo modo, se in seguito vengono introdotti altri vincoli e questi non sono soddisfatti dalla soluzione trovata in precedenza, potranno essere esplorate altre alternative per cercare una soluzione che soddisfi tutti i vincoli.

5.6.1 Implementazione del non determinismo

In JavaSet il non determinismo è stato realizzato introducendo i punti di scelta ed il backtracking. I metodi che risolvono i vincoli non deterministici sono stati implementati utilizzando il costrutto `switch` del Java.

Se un vincolo v ammette n soluzioni: sol_0,...,sol_{n-1}, allora il metodo che risolve questo tipo di vincolo avrà un'istruzione `switch` contenente `n` blocchi case: uno per ciascuna delle n alternative.

L'istruzione `switch` di un metodo non deterministico è strutturata come segue:

```
switch(v.caseControl){
   case 0:
        crea choicePoint_1
        push choicePoint_1
        applica sol_0
        storeInvariato = false;
        return;
```

```
...
case n-2:
    crea choicePoint_{n-1}
    push choicePoint_{n-1}
    applica sol_{n-2}
    storeInvariato = false;
    return;
case n-1:
    applica sol_{n-1}
    storeInvariato = false;
    return;
```

Il valore dell'espressione `switch` è dato dal valore dell'attributo `caseControl` del vincolo che si sta risolvendo. Questo attributo è `0` per default, in questo modo la prima alternativa che viene considerata è quella la cui etichetta `case` è uguale a `0`.

Ciascun blocco `case`, ad eccezione dell'ultimo, crea un punto di scelta e lo aggiunge allo stack delle alternative, cioè all'attributo statico `alternative` della classe `Solver`.

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

```
int cont;
StatoVar statoVar;
StatoStore statoStore;
```

Nell'attributo `cont` viene memorizzata l'etichetta del blocco `case` successivo a quello che crea il punto di scelta. Nell'attributo `statoVar` vengono memorizzati i riferimenti alle variabili che, al momento della creazione del punto di scelta, non sono inizializzate. L'attributo `statoStore` serve, invece, per memorizzare lo stato dello store nel momento della creazione del punto di scelta e il vincolo a cui si riferisce il punto di scelta.

La classe `StatoVar` possiede i seguenti attributi:

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

Il primo è un vettore i cui elementi sono dei riferimenti a delle `Lvar`, gli elementi del secondo vettore sono dei riferimenti a delle `Lst` e gli elementi del terzo sono dei riferimenti a dei `Set`.

La classe `StatoStore` possiede i seguenti attributi:

```
Vector stStore;
int posizione;
```

L'attributo `stStore` serve per memorizzare un clone dello store, mentre l'attributo `posizione` serve per memorizzare la posizione nello store del vincolo che ha generato il punto di scelta.

Per creare il punto di scelta e introdurlo nello stack delle alternative viene utilizzato il metodo `aggiungi_alternativa` della classe `Solver`. Questo metodo è definito come segue:

```
public static void aggiungi_alternativa(int i,StatoVar v,StatoStore s)
{
  ChoicePoint a = new ChoicePoint(i, v, s);
  alternative.push(a);
  return;
}
```

Il metodo crea un oggetto `ChoicePoint` e lo memorizza nell'attributo `alternative` della classe `Solver` che è un oggetto di tipo `Stack`. Il parametro `v` è determinato dal metodo statico della classe `Solver` `memorizzaStatoVar`, il parametro `s` è invece determinato dal metodo statico della classe `Solver` `memorizzaStatoStore`.

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

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

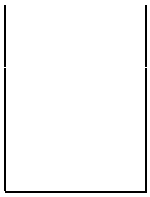
Consideriamo un semplice esempio in cui vengono risolti alcuni vincoli non deterministci e viene applicato il backtracking. In particolare vengono introdotti un vincolo di appartenenza di una variabile ad un insieme e un vincolo di uguaglianza tra due insiemi. Gli algoritmi utilizzati per riscrivere in forma risolta questi vincoli verranno descritti in dettaglio nella sezione 5.7.

Esempio 46

```
class Backtracking {
  public static void main (String[] args) throws Fallimento {
    int[] ar1 = {1,2,3};
    Set s = new Set(ar1);      // s = {1,2,3}
    int[] ar2 = {1,2};
    Set r = new Set(ar);       // r = {1,2}
    Lvar X = new Lvar();       // X = Lvar non inizializzata
    Lvar Y = new Lvar();       // Y = Lvar non inizializzata
    Lvar Z = new Lvar();       // Z = Lvar non inizializzata
    Lvar W = new Lvar();       // W = Lvar non inizializzata
    Set T = new Set(Set.vuoto.ins(Z).ins(Y).ins(X));
                               // T = {X,Y,Z}
    Solver.add(W.in(r));       // vincolo di appartenenza
    Solver.add(T.eq(s));       // vincolo di appartenenza
    Solver.add(X.neq(Z));      // vincolo di disuguaglianza
    Solver.solve();
    return;
  }
}
```

Il problema consiste nel determinare una soluzione per i vincoli $W \in \{1,2\} \wedge \{X,Y,Z\} = \{1,2,3\} \wedge X \neq W$ con X, Y, Z, W variabili logiche non inizializzata.

Lo stato iniziale dello stack delle alternative e dello store può essere rappresentato come segue.



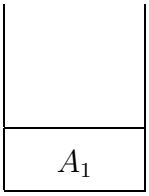
stack

$W \in \{1,2\}$	$\{X,Y,Z\} = \{1,2,3\}$	$X \neq W$

store

La risoluzione del vincolo $W \in \{1,2\}$ comporta una scelta non deterministica, in quanto è soddisfatto sia con $W = 1$ che con $W = 2$. Quindi il risolutore procede scegliendo una delle due alternative e creando un punto di scelta A_1 che viene introdotto nello stack.

Quindi lo stack delle alternative e lo store vengono modificati come segue.

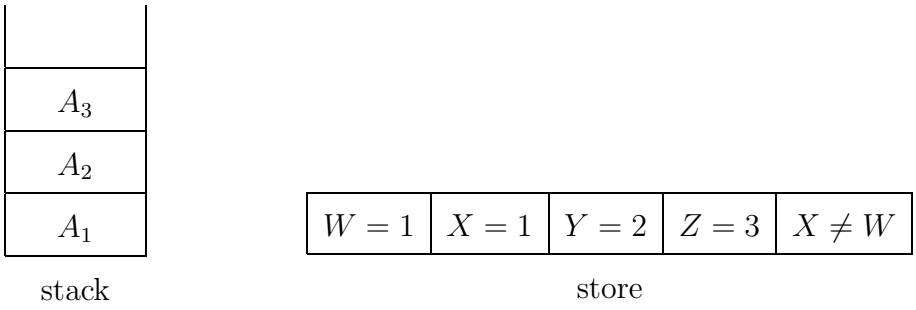


stack

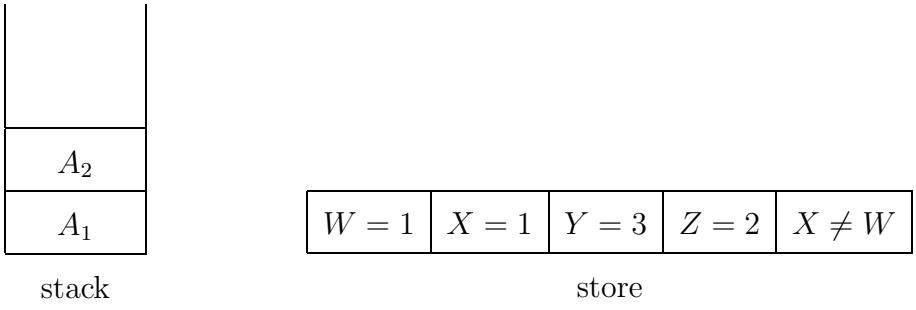
$W = 1$	$\{X,Y,Z\} = \{1,2,3\}$	$X \neq W$

store

Anche il vincolo $\{X,Y,Z\} = \{1,2,3\}$ ammette diverse soluzioni; una soluzione è data da $X = 1 \wedge \{Y,Z\} = \{2,3\}$. Quindi viene aggiunta un'altra alternativa allo stack. La risoluzione del vincolo $\{Y,Z\} = \{2,3\}$ comporta poi un'altra scelta non deterministica, in quanto ammette due soluzioni. A questo punto quindi possiamo schematizzare lo stack e lo store come segue.

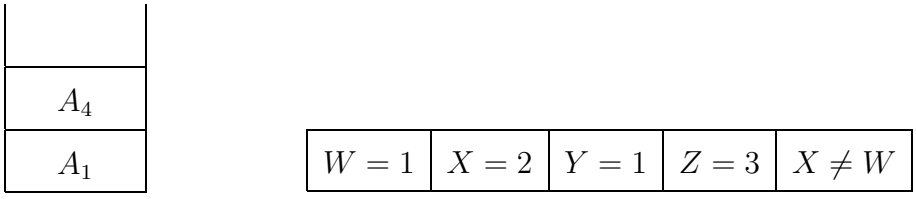


La risoluzione del vincolo $W \neq X$ porta ad un fallimento, in quanto X e W a questo punto sono entrambe uguali a 1. Quindi viene applicato il backtracking, cioè viene estratto un punto di scelta dallo stack e viene così esplorata un'altra alternativa. Nel nostro caso viene estratta l'alternativa A_3. Questa alternativa era stata generata durante la risoluzione del vincolo $\{Y, Z\} = \{2, 3\}$. Quindi lo stack e lo store vengono modificati come segue.



Lo store è ancora inconsistente, quindi viene applicato di nuovo il backtracking. Viene estratta l'alternativa A_2 e lo store e le variabili vengono riportati nelle condizioni in cui erano nel momento in cui è stato creato quel punto di scelta. Nel nostro caso viene esplorata una nuova alternativa per la risoluzione del vincolo $\{X, Y, Z\} = \{1, 2, 3\}$. Ad esempio la soluzione considerata potrebbe essere $X = 2 \wedge Y = 1 \wedge Z = 3$.

Quindi lo stack e lo store vengono modificati come segue.



Il punto di scelta A_4 viene aggiunto allo stack in quanto $\{X, Y, Z\} = \{1, 2, 3\}$ ammette altre soluzioni oltre a quelle già esplorate.

Lo store in questo caso è consistente, quindi la computazione termina. La soluzione è data da $W = 1$, $X = 2$ $Y = 1$ e $Z = 3$.

Nello stack restano memorizzati dei punti di scelta, in quanto non sono state esplorate tutte le alternative. □

5.6.2 Il metodo solve

La risoluzione dei vincoli è realizzata dal metodo `solve`. Questo metodo tiene traccia delle alternative non esplorate durante la risoluzione. Infatti i punti di scelta, creati durante l'esecuzione del metodo e non esplorati, restano memorizzati nello stack delle alternative.

Il metodo è definito come segue:

```
public static void solve() throws Fallimento {
  if(alternative.size() > 0){
     aggiornaStoreAlternative();
     aggiornaVarAlternative();
  }
  do{
     try{
         ris();
     }
     catch(Fallimento f){
         gestisciFallimento();
     }
  }
```

```
        while(!storeInvariato);
        storeSize = store.size();
        return;
    }
```

I metodi `aggiornaStoreAlternative()` e `aggiornaVarAlternative()` vengono eseguiti solo nel caso in cui lo stack delle alternative non sia vuoto. Quindi non vengono eseguiti quando il metodo `solve` viene invocato per la prima volta, in quanto in quel caso non è ancora stato creato nessun punto di scelta e quindi lo stack delle alternative è vuoto. Al contrario, i metodi `aggiornaStoreAlternative()` e `aggiornaVarAlternative()` vengono eseguiti se il metodo `solve` viene invocato dopo una risoluzione precedente che ha lasciato delle alternative inesplorate nello stack.

Questi metodi aggiornano gli attributi `statoStore` e `StatoVar` dei punti di scelta presenti nello stack delle alternative: il metodo `aggiornaStoreAlternative()` aggiunge ai vincoli dell'attributo `statoStore`, tutti i vincoli introdotti nello store dopo l' ultima risoluzione. Il metodo `aggiornaVarAlternative()` aggiunge alle variabili dell'attributo `StatoVar` le variabili non inizializzate create dopo l'ultima risoluzione.

Senza questi aggiornamenti, durante il backtracking si potrebbero perdere dei vincoli e potrebbe accadere che alcune variabili non vengano riportate allo stato corretto.

Supponiamo, ad esempio, che nello stack siano memorizzati i punti di scelta $c_1, ..., c_n$, e che dopo l'ultima invocazione della `solve` siano stati aggiunti allo store i vincoli $v_1, ..., v_p$ e siano state definite le variabili logiche non inizializzate $l_1, ..., l_r$. Nel caso in cui il metodo `solve` trovi una soluzione senza esplorare le alternative $c_1, ..., c_n$, risulta ininfluente aver aggiornato o no i punti di scelta $c_1, ..., c_n$. Al contrario, se durante il backtracking si risalisse nello stack fino a raggiungere il punto di scelta c_n (o quelli precedenti), allora lo store e le variabili verrebbero riportate nello stato in cui erano al momento della creazione di c_n. Al momento della creazione di c_n i vincoli $v_1, ..., v_p$ non erano ancora stati introdotti nello store e le variabili $l_1, ..., l_r$ non

erano ancora state create, quindi in questo caso i nuovi vincoli $v_1, ..., v_n$ verrebbero persi e le variabili $l_1 ... l_r$ non verrebbero modificate dal backtracking. Quindi, si otterrebbe una soluzione errata.

Il metodo `solve` esegue poi un ciclo `do-while` all'interno del quale viene eseguito un blocco `try-catch`. Il ciclo ricerca un punto fisso, infatti termina quando lo store non viene più modificato dal ciclo stesso, cioè quando tutti i vincoli presenti nello store sono stati riscritti in forma risolta. Il metodo `solve`, prima di terminare, aggiorna l'attributo statico `storeSize` della classe `Solver` assegnandogli l'intero `store.size()`. L'attributo `storeSize` serve per memorizzare il numero di vincoli presenti nello store al termine della risoluzione e viene utilizzato dal metodo `aggiornaStoreAlternative`.

Il costrutto `try` che si trova all'interno del ciclo `do-while` del metodo `solve` racchiude l'invocazione al metodo `ris()`. Questo metodo viene eseguito fino a quando termina con successo o solleva un'eccezione `Fallimento`.

Il metodo `ris` è definito come segue:

```
private static void ris()throws Fallimento
{
  storeInvariato = true;
  int i = 0;
   do{
     risElem(i);
     i++;
   }while(i < store.size());
  return;
}
```

Il metodo scorre tutto lo store e invoca su ciascun vincolo il metodo `risElem`. Questo metodo tenta di riscrivere in forma risolta il vincolo che gli viene passato come parametro.

Il metodo `risElem` è definito come segue :

```
private static void risElem(int k)throws Fallimento
```

```
{
 StoreElem s = (StoreElem)store.get(k);
 switch(s.cons){
    case 0: eq(s);break;
    case 1: neq(s);break;
    case 2: in(s);break;
    case 3: nin(s);break;
    case 4: disj(s);break;
    case 5: ndisj(s);break;
    case 6: union(s);break;
    case 7: nunion(s); break;
    case 8: ge(s); break;
    case 81:gt(s);break;
    case 9: le(s);break;
    case 91:lt(s);break;
    case 10:sum(s);break;
    case 11:sub(s);break;
    case 12:mul(s);break;
    case 13:div(s);break;
    case 14:less(s);break;
 }
 return;
}
```

Questo metodo valuta di che tipo è il vincolo che gli è stato passato come parametro e sceglie il metodo adatto alla risoluzione di quel vincolo. Supponiamo ad esempio che si tratti di un vincolo di uguaglianza, in questo caso quindi viene invocato il metodo statico `eq`.

Il metodo `eq` è definito come segue:

```
private static void eq(StoreElem s)throws Fallimento
{
 if(s.obj1 instanceof Lvar && s.obj2 instanceof Lvar)
      eqLvar((Lvar)s.obj1,(Lvar)s.obj2,s);  // X.eq(Y)  entrambe Lvar
 else if(s.obj1 instanceof Lvar && s.obj2 instanceof Lst)
      eqLvarLst((Lvar)s.obj1,(Lst)s.obj2,s);// X.eq([t1,..,tn])
 else if(s.obj1 instanceof Lst && s.obj2 instanceof Lvar)
      eqLvarLst((Lvar)s.obj2,(Lst)s.obj1,s);// [t1,..,tn].eq(X)
```

```
  else if(s.obj1 instanceof Lst && s.obj2 instanceof Lst)
       eqLst((Lst)s.obj1,(Lst)s.obj2,s);     // [t1,..,tn].eq([s1,..,sn])
  else if(s.obj1 instanceof Lvar && s.obj2 instanceof Set)
       eqLvarSet((Lvar)s.obj1,(Set)s.obj2,s);// X.eq({...})
  else if(s.obj1 instanceof Set && s.obj2 instanceof Lvar)
       eqLvarSet((Lvar)s.obj2,(Set)s.obj1,s);// {...}.eq(X)
  else if(s.obj1 instanceof Set && s.obj2 instanceof Set)
       eqSet((Set)s.obj1,(Set)s.obj2,s);     // {...}.eq({...})
  else if(s.obj1 instanceof Lvar)
       eqLvarObj((Lvar)s.obj1,s.obj2,s);     // X.eq(t)
  else if(s.obj2 instanceof Lvar)
       eqLvarObj((Lvar)s.obj2,s.obj1,s);     // t.eq(X)
  else if(s.obj1 instanceof Lst)
       eqLstObj((Lst)s.obj1,s.obj2,s);       // [t1,..,tn].eq(t)
  else if(s.obj2 instanceof Lst)
       eqLstObj((Lst)s.obj2,s.obj1,s);       // t.eq([t1,..,tn])
  else if(s.obj1 instanceof Set)
       eqSetObj((Set)s.obj1,s.obj2,s);       // {...}.eq(t)
  else if(s.obj2 instanceof Set)
       eqSetObj((Set)s.obj2,s.obj1,s);       // t.eq({...})
  else eqObj(s.obj1,s.obj2,s);               // t.eq(s)
   return;
 }
```

Il metodo valuta di che tipo sono gli oggetti coinvolti nel vincolo e poi esegue il metodo adatto. Ad esempio, se si tratta di un vincolo di uguaglianza tra due insiemi, viene eseguito il metodo `eqSet`.

Se un vincolo è inconsistente, viene sollevata un'eccezione `Fallimento`, che risale la catena delle chiamate fino a raggiungere il metodo `solve` dove viene catturata. La clausola `catch` che gestisce l'eccezione invoca il metodo `gestisciFallimento()` che è realizzato come segue:

```
  private static void gestisciFallimento()throws Fallimento
  {
   try{
       ChoicePoint c = estrai_alternativa();
       ripristinaStatoVar(c.statoVar);
```

```
        ripristinaStatoStore(c.statoStore.statoStore);
        ((StoreElem)c.statoStore.statoStore.get(c.statoStore.posizione)).
          caseControl = c.cont;
        storeInvariato = false;
        return;
    }
    catch(EmptyStackException e){
      throw new NoSolutionFound();
    }
  }
```

Il metodo `gestisciFallimento` estrae dallo stack il punto di scelta `c` mediante il metodo `estrai_alternativa()`. Questo metodo utilizza il metodo `pop` della classe `Stack`, cioè estrae il punto di scelta introdotto per ultimo.

I metodi `ripristinaStatoVar` e `ripristinaStatoStore` riportano le variabili (`Lvar`, `Lst`, `Set`) e lo store nello stato in cui erano quando è stato creato quel punto di scelta. Infine viene modificato il campo `caseControl` del vincolo che ha generato il punto di scelta in modo che, nella valutazione successiva di quel vincolo, verrà eseguito il blocco `case` successivo a quello che ha causato il fallimento, cioè verrà esplorata un'altra alternativa.

Se al momento dell'estrazione dell'alternativa lo stack è vuoto, viene sollevata un'eccezione di tipo `EmptyStackException`, in questo caso non ci sono più altre alternative, quindi viene sollevata un'eccezione `NoSolutionFound` e la computazione termina.

5.7 Procedure di riscrittura

Nel corso di questo capitolo abbiamo parlato delle procedure di riscrittura per i vincoli e di come queste procedure in molti casi portino a scelte non deterministiche. In questa sezione, per chiarire questi concetti, descriveremo ad esempio le procedure di riscrittura per i vincoli di uguaglianza e di appartenenza.

In figura 5.1 è riportata una descrizione astratta dell'algoritmo utilizzato per

riscrivere in forma risolta i vincoli di uguaglianza. Questo algoritmo è implementato dal metodo `eq` della classe `Solver` che è descritto nella sezione precedente.

$eq(C)$: regole da applicare a C fino a quando C non è in forma risolta		
(1)	$X = X \wedge C'$	$\longmapsto C'$
(2)	$t = X \wedge C'$ t non è una variabile	$\longmapsto X = t \wedge C'$
(3)	$X = [t_0, ..., t_n] \wedge C'$ $X \in vars(t_0, ..., t_n)$	$\longmapsto$ fail
(4)	$X = \{t_0, ..., t_n \mid t\} \wedge C'$ t è $\emptyset$ o una variabile, $X \in vars(t_0, ..., t_n)$	$\longmapsto$ fail
(5)	$X = t \wedge C'$ $X \notin vars(t)$, t insieme o X insieme $\notin C'$	$\longmapsto X = t \wedge C'[X/t]$
(6)	$X = \{t_0, ..., t_n \mid X\} \wedge C'$ $X \notin vars(t_0, ..., t_n)$,	$\longmapsto X = \{t_0, ..., t_n \mid N\} \wedge C'$
(7)	$[s_0, ..., s_n] = [t_0, ..., t_m] \wedge C'$	$\longmapsto$ fail
(8)	$[s_0, ..., s_n] = [t_0, ..., t_n] \wedge C'$	$\longmapsto s_1 = t_1 \wedge ... \wedge s_n = t_n \wedge C'$
(9)	$\{t \mid s\} = \{t' \mid s'\} \wedge C'$	$\longmapsto C' \wedge$ una delle seguenti: (i) $t = t' \wedge s = s'$ (ii) $t = t' \wedge \{t \mid s\} = s'$ (iii) $t = t' \wedge s = \{t' \mid s'\}$ (iv) $s = \{t' \mid N\} \wedge \{t \mid N\} = s'$
(10)	$\{t_0, ..., t_m \mid X\} = \{t'_0, ..., t'_n \mid X\} \wedge C'$	$\longmapsto C' \wedge$ una delle seguenti: (i) $t_0 = t'_j \wedge \{t_1, ..., t_m \mid X\} = \{t'_0, ..., t'_{j-1}, t'_{j+1}, ..., t'_n \mid X\}$ (ii) $t_0 = t'_j \wedge \{t_0, ..., t_m \mid X\} = \{t'_0, ..., t'_{j-1}, t'_{j+1}, ..., t'_n \mid X\}$ (iii) $t_0 = t'_j \wedge \{t_1, ..., t_m \mid X\} = \{t'_0, ..., t'_n \mid X\}$ (iv) $X = \{t_0 \mid N\} \wedge \{t_1, ..., t_m \mid N\} = \{t'_0, ..., t'_n \mid N\}$ per ogni j in $\{0, ..., n\}$

Figura 5.1: regole di riscrittura per i vincoli di uguaglianza

Come si può osservare la risoluzioni dei vincoli di uguaglianza in alcuni casi è deterministica, in altri invece è non deterministica.

Consideriamo, ad esempio, il seguente vincolo di uguaglianza tra insiemi:

$$\{X|R\} = \{Y|S\}$$

X e Y sono elementi qualunque, R ed S sono insiemi. Applicando la regola (9), l'algoritmo di unificazione determina quattro soluzioni:

$$X = Y, R = S$$

$$X = Y, S = \{X|R\}$$

$$X = Y, R = \{Y|S\}$$

$$R = \{Y|N\}, S = \{X|N\}$$

In JavaSet questa parte dell'algoritmo è implementato come segue:

```
public static void eqSet(Set set1, Set set2, StoreElem s)
throws Fallimento
{
 ...
 switch(s.caseControl){

  case 0:                              // X = Y, R = S
    aggiungi_alternativa(1, (StatoVar)memorizzaStatoVar(),
                            (StatoStore)memorizaStatoStore(s));
    s.obj1 = set1.first();
    s.obj2 = set2.first();
    StoreElem se0 = new StoreElem(set1.sub(), 0,set2.sub());
    store.add(store.indexOf(s)+1,se0);
    eq(s);
    storeInvariato = false;
    return;

  case 1:                              // X = Y, S = {X|R}
    aggiungi_alternativa(2, (StatoVar)memorizzaStatoVar(),
                            (StatoStore)memorizaStatoStore(s));
    s.obj1 = set1.first();
    s.obj2 = set2.first();
```

```
    s.caseControl = 0;
    StoreElem se1 = new StoreElem(set1, 0, set2.sub());
    store.add(store.indexOf(s)+1,se1);
    eq(s);
    storeInvariato = false;
    return;

  case 2:                              // X = Y, R = {Y|S}
    aggiungi_alternativa(3, (StatoVar)memorizzaStatoVar(),
                            (StatoStore)memorizaStatoStore(s));
    s.obj1 = set1.first();
    s.obj2 = set2.first();
    s.caseControl = 0;
    StoreElem se2 = new StoreElem(set1.sub(), 0, set2);
    store.add(store.indexOf(s)+1,se2);
    eq(s);
    storeInvariato = false;
    return;

  case 3:                              // R = {Y|N}, S = {X|N}
    Set n = new Set();
    s.obj1 = set1.sub();
    s.obj2 = n.ins(set2.first());
    s.caseControl = 0;
    StoreElem se3 = new StoreElem(n.ins(set1.first()), 0, set2.sub());
    store.add(store.indexOf(s)+1,se3);
    eq(s);
    storeInvariato = false;
    return;
}
...
```

L'istruzione `switch` in questo caso prevede quattro blocchi `case`, uno per ciascuna delle quattro alternative. Il primo `case` sostituisce il vincolo $\{X|R\}$ con i vincoli $X = Y$ e $R = S$, il secondo lo sostituisce con i vincoli $X = Y$ e $S = \{X|R\}$, il terzo con i vincoli $X = Y$ e $R = \{Y|S\}$ e il quarto con i vincoli $R = \{Y|N\}, S = \{X|N\}$.

In figura 5.2 è riportata una descrizione astratta dell'algoritmo utilizzato per riscrivere in forma risolta i vincoli di appartenenza. Questo algoritmo è implementato dal metodo `in`.

Come si può osservare, anche la risoluzione dei vincoli di appartenenza in un caso è non deterministica.

$in(C)$: regole da applicare a C fino a quando C non è in forma risolta		
(1)	$s \in \emptyset \wedge C'$	$\longmapsto fail$
(2)	$r = \{s\|t\} \wedge C'$	$\longmapsto C' \wedge$ una delle seguenti: $(i) \quad r = s$ $(ii) \quad r \in t$
(3)	$t \in X \wedge C'$	$\longmapsto X = \{t\|N\} \wedge C'$

Figura 5.2: procedure di riscrittura per i vincoli di appartenenza

Supponiamo, ad esempio, di dover risolvere il seguente vincolo di appartenenza:

$$r \in \{s|t\}$$

con r e s elementi qualunque e t istanza della classe `Set`. Applicando la regola (2), l'algoritmo determina due soluzioni:

$$r = s$$

$$r \in t$$

In JavaSet questo algoritmo è implementato come segue:

```
public static void inLvarSet(Lvar lvar, Set set, StoreElem s)
throws Fallimento
{
 ...
  switch(s.caseControl){
    case 0:                              // X.eq(t)
```

```
      StatoVar statoVarIn = memorizzaStatoVar();
      StatoStore statoStoreIn = memorizzaStatoStore(s);
      if(set.size() > 1 || !set.ultimoResto().iniz)
         aggiungi_alternativa(1, statoVarIn, statoStoreIn);
      s.obj1 = lvar;
      s.cons = 0;
      s.obj2 = set.first();
      eq(s);
      storeInvariato = false;
      return;
    case 1:                              // X.in(R)
      s.caseControl = 0;
      s.obj2 = (Set)set.sub();
      in(s);
      storeInvariato = false;
      return;
  }
 ...
}
```

In questo caso l'istruzione `switch` prevede due blocchi `case`: il primo blocco sostituisce il vincolo $r \in \{s|t\}$ con il vincolo $r = s$, il secondo lo sostituisce con il vincolo $r \in t$.

I vincoli di uguaglianza, appartenenza, unione, disgiunzione e le loro negazioni sono definiti come vincoli "primitivi". Per ogniuno di essi esiste un costruttore della classe `StoreElem` che permette di costruire un apposito elemento dello store. Il metodo `risElem` (cfr. sezione 5.6.2 pag. 131) prevede poi un metodo per la risoluzione di ciascuno di questi vincoli.

Tramite questi vincoli abbiamo poi definito il vincolo `less`, i vincoli di inclusione, intersezione, differenza insiemistica e le loro negazioni, essendo valide le seguenti equivalenze:

$$
\begin{array}{rll}
s \subseteq t & \text{sse} & t = t \cup s \\
s \not\subseteq t & \text{sse} & t \neq t \cup s \\
t = r \cap s & \text{sse} & \exists R, S(r = R \cup t \wedge s = S \cup t \wedge R||S) \\
t \neq r \cap s & \text{sse} & \exists T, (T = r \cap s \wedge T \neq t) \\
t = r \setminus s & \text{sse} & \exists W, (r = t \cup r \wedge W = s \cup t \wedge W = r \cup W \wedge s||t) \\
t \neq r \setminus s & \text{sse} & \exists T, (T = r \setminus s \wedge T \neq t) \\
S.less(X, R) & \text{sse} & S = \{X|R\} \wedge X \notin R
\end{array}
$$

Quindi l'introduzione di uno di questi vincoli nello store avviene introducendo una congiunzione di vincoli "primitivi".

Ad esempio il metodo `inters` della classe `Set` è definito come segue:

```
public Vector inters(Set r,Set s)
{
 if(this.equ == null && r.equ == null && s.equ == null){
    Vector v = new Vector();
    Set R = new Set();
    Set S = new Set();
    v.add(r.union(R,this));
    v.add(s.union(S,this));
    v.add(R.disj(S));
    return v;
 }
 else{
    if(this.equ != null)return this.equ.inters(r,s);
    else if(r.equ != null)return this.inters(r.equ,s);
    else return this.inters(r,s.equ);
 }
}
```

$t.inters(r, s)$ restituisce il vincolo $r.union(R, t) \wedge s.union(S, t) \wedge R.disj(S)$ con R e S insiemi non inizializzati costruiti dalla `inters` stessa. Quindi l'espressione $Solver.add(t.inters(r, s))$ aggiunge allo store due vincoli di unione e uno di disgiunzione.

Il discorso è del tutto analogo per gli altri vincoli. Ad esempio l'espressione $Solver.add(s.subset(t))$ aggiunge allo store il vincolo $s.union(t, s)$, infatti il metodo `subset` della classe `Set` restituisce un oggetto di tipo `StoreElem` che il metodo

`add` aggiunge allo store. L'espressione $Solver.add(S.less(X,T))$ aggiunge allo store i vincoli $S.eq(T.ins(X))$ e $X.nin(T)$. L'espressione $Solver.add(t.differ(r,s))$ aggiunge allo store il vincolo $r.union(t,r) \wedge W.union(s,t) \wedge W.union(r,W) \wedge s.disj(t)$ con W insieme non inizializzato.

5.8 Ricerca di una o di tutte le soluzione

Oltre al metodo `solve` che permette di determinare una soluzione tenendo traccia delle alternative rimaste inesplorate, in JavaSet è anche possibile determinare una soluzione per i vincoli dati ed eliminare tutte le altre alternative, oppure determinare tutte le possibili soluzioni per una variabile, in modo che sia soddisfatto l'insieme dei vincoli dato.

Il metodo statico `solve1` determina una soluzione per l'insieme dei vincoli dati utilizzando il metodo `solve`, ma non tiene traccia delle alternative rimaste inesplorate, quindi in caso di backtracking, quelle alternative non vengono considerate.

Il metodo `solve1` è realizzato come segue:

```
public static void solve1() throws Fallimento
{
 solve();
 alternative.removeAllElements();
 return;
}
```

Il metodo statico `setof`, permette invece di determinare tutte le soluzioni per una variabile. Al termine dell'esecuzione di questo metodo, lo stack delle alternative è vuoto, in quanto il metodo non lascia nessuna alternativa inesplorata.

Il metodo è stato sovraccaricato in modo da poter ricevere come parametro una `Lvar`, una `Lst` o un `Set`.

L'implementazione è la seguente:

```
 public static Lst setof(Set s)throws Fallimento
 {
   Vector TuttiValori = new Vector();
   solve();
   Set news = s.clona();
   TuttiValori.add(news);
   while(alternative.size() != 0){
         gestisciFallimentoSetof();
         if(solveSetof()){
            Set news1 = s.clona();
            TuttiValori.add(news1);
         }
   }
   return new Lst(TuttiValori, Lst.vuoto);
 }
```

Il metodo `setof` definisce il vettore `TuttiValori` nel quale vengono memorizzati tutti i valori che la variabile può assumere. La prima soluzione viene trovata applicando il metodo `solve`. Se i vincoli sono inconsistenti la `solve` genera un eccezione `NoSolutionFound` e la computazione termina, altrimenti il metodo esegue un ciclo `while`, che termina quando lo stack è vuoto, cioè quando sono state esplorate tutte le alternative. All'interno del ciclo viene forzato un fallimento mediante il metodo `gestisciFallimentoSetof`. Il metodo è definito come segue:

```
public static void gestisciFallimentoSetof()throws Fallimento
 {
  try{
     ChoicePoint c = estrai_alternativa();
     ripristinaStatoVar(c.statoVar);
     ripristinaStatoStore(c.statoStore.statoStore);
     ((StoreElem)c.statoStore.statoStore.get(c.statoStore.posizione)).
       caseControl = c.cont;
     storeInvariato = false;
     return;
  }
```

```
  catch(EmptyStackException e){
     return;
  }
 }
```

Questo metodo è del tutto simile al metodo `gestisciFallimento` ma in questo caso la clausola `catch` che cattura l'eccezione `EmptyStackException` esegue l'istruzione `return`, cioè fa tornare l'esecuzione al metodo chiamante, invece la clausola `catch` del metodo `gestisciFallimento` solleva un'eccezione `NoSolutionFound` e quindi fa terminare l'esecuzione.

Dopo il metodo `gestisciFallimentoSetof`, viene eseguita un'istruzione `if` il cui argomento è il valore booleano restituito dal metodo `solveSetof` della classe `Solver`. Se il metodo restituisce il valore `true`, viene aggiunto al vettore `TuttiValori` il nuovo valore trovato dal metodo `solveSetof` stesso.

Il metodo `solveSetof` è definito come segue:

```
  private static boolean solveSetof() throws Fallimento
  {
  if(alternative.size() > 0){
     aggiornaStoreAlternative();
     aggiornaVarAlternative();
  }
  do{
     try{
         ris();
     }
     catch(Fallimento f){
         if(alternative.size() > 0)
            gestisciFallimentoSetof();
         else return false;
     }
  }
  while(!storeInvariato);
  storeSize = store.size();
  return true;
  }
```

Questo metodo si differenzia dal metodo `solve` solo nel blocco `catch`: la clausola `catch` del metodo `solve` esegue il metodo `gestisciFallimento`, mentre quella del metodo `solveSetof`, se ci sono ancora dei punti di scelta nello stack, esegue il metodo `gestisciFallimentoSetof`, altrimenti restituisce il valore `false`. Se la `solveSetof` restituisce il valore `false`, il metodo `setof` si interrompe restituendo una lista i cui elementi sono gli elementi del vettore `TuttiValori`.

5.9 Il metodo forall

Il metodo statico `forall` realizza una struttura di controllo di tipo `for`, cioè permette di applicare uno o più vincoli a ciascun elemento di un insieme. L'espressione $Solver.forall(X, S, v_X)$ in termini astratti equivale all'espressione $\forall X \in S$, v_X , dove v_X è un vincolo contenente X.

Il metodo `forall` è stato sovraccaricato in modo da poter ricevere come terzo parametro sia un oggetto di tipo `StoreElem`, cioè un singolo vincolo, sia un oggetto di tipo `VettStoreElem` cioè una congiunzione di vincoli. Nel primo caso aggiunge allo store un solo oggetto `StoreElem`, mentre nel secondo caso aggiunge alo store tutti i vincoli della congiunzione.

Il metodo è realizzato come segue:

```
public static void forall(Lvar X,Set s,StoreElem st)
throws Fallimento
{
 if(s.isEmpty())return;
 else{
   Set R = new Set();
   add(s.less(X,R));
   add(st);
   Lvar Y = new Lvar();
   StoreElem stnew = new StoreElem(st.sost(X,Y));
   solve();
   forall(Y,R,stnew);
```

```
    return;
  }
 }
```

Il metodo utilizza il vincolo `less` per estrarre dal `Set s` la `Lvar X` e poi introduce il vincolo `st` sulla variabile estratta aggiungendo `st` allo store. Il vincolo `st` deve però essere soddisfatto da tutti gli elementi dell'insieme `s`, quindi viene creata una nuova variabile `Y` e un nuovo oggetto `StoreElem` ottenuto da `st` sostituendo `X` con `Y` ovunque compaia nel vincolo e poi viene fatta una chiamata ricorsiva al metodo `forall`. Prima di eseguire di nuovo il `forall` è però necessario risolvere il vincolo `less` in modo che al passo successivo l'insieme `R`, ottenuto da `s` estraendo `X`, sia inizializzato. Se `R` è l'insieme vuoto la ricorsione termina.

Capitolo 6

Esempi

In questo capitolo descriveremo alcuni semplici programmi d'esempio che mostrano in che modo si può utilizzare la libreria JavaSet per risolvere problemi di soddisfacimento di vincoli. In particolare tratteremo il problema delle n-regine, il problema del commesso viaggiatore e il problema della colorazione di una mappa. Sono problemi ben noti e che vengono spesso affrontati nella programmazione logica con vincoli. Nella risoluzione di questi problemi l'utilizzo dei vincoli su insiemi risulta molto naturale.

6.1 Problema delle n-regine

Il problema è descritto come segue: Su una scacchiera $N \times N$ si richiede di posizionare N regine in modo tale che non si attacchino a vicenda, i.e., due regine non possono trovarsi sulla stessa riga, sulla stessa colonna o sulla stessa diagonale.

Per modellare il problema si può osservare che in qualunque soluzione si avrà una regina su ciascuna riga ed una su ciascuna colonna. Possiamo quindi assegnare una variabile logica a ciascuna riga. Per fare questo possiamo costruire una lista di N variabili logiche $[r_0, ..., r_{N-1}]$. La prima variabile della lista è associata alla prima riga, la seconda è associata alla seconda riga ecc. Per ogni riga ci sono N colonne su cui possiamo piazzare la regina assegnata a quella riga. Se identifichiamo le colonne con gli interi $0, ..., N-1$, la risoluzione del problema consiste nell'assegnare

a ciascuna delle N variabili logiche un valore tra 0 e $N-1$ in modo che per ogni $i \neq j$, con i e j compresi tra 0 e $N-1$, siano soddisfatti i seguenti vincoli:

$$r_i \neq r_j$$
$$r_j + j - r_i \neq i$$
$$r_i + j - r_j \neq i$$

Il primo vincolo assicura che due regine non si trovino sulla stessa colonna, gli altri due assicurano che due regine non si trovino sulla stessa diagonale.

Di seguito riportiamo il programma per la risoluzione del problema delle N-regine, per $N = 4$ scritto utilizzando JavaSet:

```
import JavaSet.*;
import java.io.*;

class Queens
{
 public static final int N = 4;
 public static Set board = new Set(0,N-1);   // Set board = {0,..,N-1}
 public static Lst pos = Lst.mkLst(N);       // Lst di N Lvar


public static void main (String[] args) throws IOException,Fallimento
{
 for(int i = 0; i < N; i++){
     Solver.add(((Lvar)pos.get(i)).in(board));
 }
 Solver.add(Solver.allDifferent(pos));
 Solver.solve();

 for(int i = 0; i < N-1; i++){
  for(int j = i+1; j < N; j++){
   Solver.add(((Lvar)pos.get(j)).sum(j).sub((Lvar)pos.get(i)).neq(i));
   Solver.add(((Lvar)pos.get(i)).sum(j).sub((Lvar)pos.get(j)).neq(i));
  }
 }
 Solver.solve();
```

```
  for(int i = 0; i < N; i++){              // stampa della soluzione
      System.out.print("\nQueen "+i+" = ");
      Lvar.output((Lvar)pos.get(i));
  }
  return;
 }
}
```

`N` è definito come una costante, che in questo caso è posta uguale a `4`. L'insieme `board` è dato dagli interi contenuti nell'intervallo $0 \div (N-1)$, questo insieme indica le colonne su cui possono essere piazzate le regine. La lista `pos` serve ad indicare le posizioni delle regine: il valore dell'i-esima `Lvar` della lista `pos` indica la colonna in cui è piazzata la regina assegnata alla i-esima riga.

Su ciascuna variabile logica della lista `pos` abbiamo introdotto il vincolo si appartenenza all'insieme `board`, in modo da assegnare una colonna a ciascuna variabile. Per assicurarci che su ciascuna colonna sia piazzata una sola regina abbiamo utilizzato il metodo `allDifferent`, con la lista `pos` come parametro.

I vincoli:

$$r_j + j - r_i \neq i$$
$$r_i + j - r_j \neq i$$

sono stati introdotti mediante le istruzioni seguenti:

```
Solver.add(((Lvar)pos.get(j)).sum(j).sub((Lvar)pos.get(i)).neq(i));
Solver.add(((Lvar)pos.get(i)).sum(j).sub((Lvar)pos.get(j)).neq(i));
```

in cui `pos.get(i)` e `pos.get(j)` sono, rispettivamente, l'i-esima e la j-esima `Lvar` della lista `pos`.

L'output del programma, rappresentante una delle possibili soluzioni del problema, è il seguente:

```
Queen 0 = 1
Queen 1 = 3
Queen 2 = 0
Queen 3 = 2
```

La soluzione trovata corrisponde alla disposizione mostrata in figura 6.1.

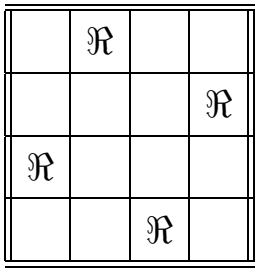


Figura 6.1: Una soluzione del problema delle n-regine, con $n = 4$.

6.2 Problema del commesso viaggiatore

Un altro esempio dell'impiego dei vincoli su insiemi è fornito dalla risoluzione del problema del commesso viaggiatore (TSP: Traveling Salesman Problem).

Dato l'insieme dei nodi N e il grafo G degli archi percorribili, il problema consiste nel determinare, se esiste, un percorso che permetta di partire da un nodo assegnato appartenente a N e di ritornare in tale punto passando una ed una sola volta per tutti gli altri nodi e percorrendo archi appartenenti a G.

Il grafo G, dagli archi percorribili, può essere realizzato utilizzando un insieme i cui elementi sono liste di coppie di nodi. Se il grafo contiene la lista $[a, b]$ significa che dal nodo a si può raggiungere il nodo b. La scelta di utilizzare le liste è dettata dal fatto che il grafo che vogliamo rappresentare è orientato.

Se con p indichiamo il nodo di partenza, il primo passo per risolvere il problema è quello di determinare un nodo s in modo tale che la lista $[p, s]$ appartenga al grafo G, poi si considera s come nuovo nodo sorgente e si prosegue nello stesso modo fino a visitare tutti i nodi di N.

Ogni nodo deve essere visitato una ed una sola volta, quindi ad ogni passo devono essere noti i nodi già visitati per assicurarsi che non vengano più rivisitati. Per fare questo si può definire un insieme v, inizialmente vuoto, nel quale memorizzare di

volta in volta i nodi visitati. Ad ogni passo, il nodo s che viene determinato deve soddisfare il vincolo $s \notin v$, se soddisfa questo vincolo viene aggiunto, a sua volta, all'insieme v in modo da non essere più visitato.

Il percorso svolto viene memorizzato mediante una lista L, che contiene l'elenco dei nodi visitati, nell'ordine in cui sono stati visitati. Quando la dimensione della lista P è uguale al numero dei nodi introdotti significa che sono stati visitati tutti i nodi dell'insieme N.

A questo punto resta da verificare che dall'ultimo nodo visitato si può raggiungere il nodo di partenza, cioè deve essere soddisfatto il vincolo $[u, p] \in G$ con u ultimo nodo visitato.

Il problema può essere risolto come segue:

```
import JavaSet.*;
import java.io.*;

class Tsp
{
 static int n;
 static Set nodes = Set.vuoto;
 static Set grafo = Set.vuoto;
 static Set visited = Set.vuoto;
 static Lst percorso = Lst.vuoto;
```

La classe `TSP` possiede cinque attributi statici: l'attributo `n` serve per memorizzare il numero di nodi introdotti, l'insieme `nodes` corrisponde all'insieme N, in esso verranno memorizzati i nodi, l'insieme `grafo` corrisponde a G, cioè viene utilizzato per memorizzare gli archi percorribili, l'insieme `visited` corrisponde all'insieme dei nodi visitati v e infine la lista `percorso` corrisponde alla lista P utilizzata per memorizzare il percorso svolto.

Il metodo `main` richiede di inserire i nodi gli archi e il nodo di partenza poi inizia la risoluzione. I nodi, agli archi e il nodo di partenza vengono introdotti da tastiera e letti mediante i metodi `leggiNodi` e `leggiArchi`:

```
 public static void main (String[] args) throws IOException,Fallimento
 {
  InputStreamReader reader = new InputStreamReader(System.in);
  BufferedReader input = new BufferedReader(reader);
  System.out.println();
  System.out.print("Immetti la lista dei nodi (separati da uno spazio): ");
  leggiNodi(input);
  System.out.println();
  for(int i = 0; i < n; i++){
   System.out.print("Immetti i nodi raggiungibili da "+ nodes.get(i)+" : ");
   leggiArchi(input,(String)nodes.get(i));
  }
  System.out.println();
  System.out.print("Immetti il nodo di partenza: ");
  String sorgente = input.readLine();
  tsp(nodes,grafo,sorgente);
  mostraPercorso();
  return;
 }
}

 public static void leggiNodi(BufferedReader in)
 throws IOException
 {
  String nod = in.readLine();
  for(int i = nod.length()-1; i >= 0; i--){
      if(nod.charAt(i) != ' '){
         nodes = nodes.ins(String.valueOf(nod.charAt(i)));
         n = n + 1;
      }
  }
 }

 public static void leggiArchi(BufferedReader in, String source)
 throws IOException
 {
  String archi = in.readLine();
    for(int i = archi.length()-1; i >= 0; i--){
      if(archi.charAt(i) != ' '){
```

```
            String[] arc = {source,String.valueOf(archi.charAt(i))};
            Lst arco = new Lst(arc);
            grafo = grafo.ins(arco);
        }
    }
}
```

La ricerca della soluzione inizia con l'invocazione del metodo `tsp`. Questo metodo è stato sovraccaricato con due definizioni che si differenziano per il tipo di uno dei parametri passati.

Il metodo `main` invoca il metodo `tsp` con parametri l'insieme dei nodi, il grafo e il punto iniziale. Questo metodo, che è riportato di seguito, verifica se il nodo di partenza appartiene all'insieme dei nodi, poi aggiunge il nodo all'insieme `visited` in modo da non tornare nel nodo di partenza se non dopo aver visitato tutti gli altri nodi, inoltre il nodo viene memorizzato in prima posizione nella lista `percorso`. Infine il metodo invoca l'altro metodo `tsp`.

```
public static void tsp(Set nodi,Set grafo,String source)
throws Fallimento
{
 Lvar part = new Lvar(source);
 Solver.add(part.in(nodi));
 Solver.solve();
 visited = visited.ins(part);
 percorso = percorso.ins1(part);
 tsp(grafo, part, source);
 return;
}
```

Di seguito riportiamo il metodo `tsp` con parametri un `Set`, una `Lvar` e una stringa. Questo metodo è definito in modo ricorsivo. Ad ogni passo crea una variabile non inizializzata `x` e determina il suo valore in modo che `[part,x]` sia un arco dell'insieme `grafo` e il nodo `x` non sia ancora stato visitato. Ad ogni passo deve essere invocato il metodo `solve`, in quanto al passo successivo deve essere noto il

nodo in cui ci si trova. La ricorsione termina quando tutti i nodi dell'insieme `nodes` sono stati visitati.

```
public static void tsp(Set grafo, Lvar part, String source)
throws Fallimento
{
 Lvar x = new Lvar();
 Lvar[] arc1 = {part,x};
 Lst a1 = new Lst(arc1);
 Solver.add(a1.in(grafo));
 Solver.add(x.nin(visited));
 Solver.solve();
 visited = visited.ins(x);
 percorso = percorso.insn(x);
 if(percorso.size() < n) tsp(grafo,x,source);
 else tsplast(grafo,x,source);
 return;
}
```

Al termine della ricorsione il metodo `tsp` invoca il metodo `Tsplast` che verifica se il nodo di partenza è raggiungibile dal nodo visitato per ultimo.

Questo metodo è definito come segue:

```
public static void tsplast(Set grafo,Lvar z,String source)
throws Fallimento
{
 Lst a2 = new Lst(Lst.vuoto.ins1(source).ins1(z));
 Solver.add(a2.in(grafo));
 Solver.solve();
 return;
}
```

Se il problema è soddisfacibile, l'istruzione `return` del metodo `tsp` trasferisce il controllo al metodo `main` il quale invoca il metodo `mostraPercorso()`. Questo metodo stampa la soluzione ottenuta, utilizzando la lista `percorso`.

Il metodo è definito come segue:

```
public static void mostraPercorso() throws IOException
{
 System.out.println(" ");
 System.out.print(" percorso: ");
 for(int i = 0; i < n; i++){
     Lvar.output((Lvar)percorso.get(i));System.out.print(" -> ");
 }
 Lvar.output((Lvar)percorso.get(0));
 System.out.println(" ");
 return;
}
```

Di seguito riportiamo un esempio d'esecuzione del programma:

```
Immetti la lista dei nodi (separati da uno spazio): a b c d e

Immetti i nodi raggiungibili da a : b c
Immetti i nodi raggiungibili da b : a e
Immetti i nodi raggiungibili da c : b
Immetti i nodi raggiungibili da d : a
Immetti i nodi raggiungibili da e : d

Immetti il nodo di partenza: a

   percorso: a -> c -> b -> e -> d -> a
```

6.3 Problema della colorazione di una mappa

Il problema della colorazione di una mappa geografica è definito come segue: Data una mappa di n regioni r_1,...,r_n e un insieme di colori $C = \{c_1, ...c_m\}$ bisogna assegnare un colore a ciascuna regione della mappa in modo che regioni confinanti abbiano colori diversi.

Le n regioni possono essere rappresentate con n `Lvar` ciascuna delle quali verrà inizializzata con un colore dell'insieme C. La mappa può essere considerata come un grafo non orientato in cui i nodi sono le regioni e gli archi sono le coppie di regioni

confinanti, quindi possiamo rappresentarla come un insieme i cui elementi sono insiemi contenenti due regioni ciascuno. Ad esempio, se l'insieme $\{r_i, r_j\}$ appartiene alla mappa, significa che le regioni r_i ed r_j sono confinanti.

Per determinare una soluzione bisogna assegnare a ciascuna regione un colore in modo che, per ogni arco $\{r_i, r_j\}$ della mappa e per ogni regione r_k appartenente all'insieme delle regioni, sia soddisfatto il vincolo $\{r_i, r_j\} \neq \{r_k\}$, infatti questo vincolo è soddisfatto se e solo se ad r_i e ad r_j sono stati assegnati colori diversi.

Di seguito riportiamo un programma d'esempio per la risoluzione del problema della colorazione di una mappa:

```
import java.io.*;
import JavaSet.*;

class Coloring {
 public static void coloring(Set regioni,Set mappa,Set colori)
 throws Fallimento
 {
  Lvar Y = new Lvar();
  Solver.add(regioni.eq(colori));
  Solver.forall(Y,regioni,(Set.vuoto.ins(Y)).nin(mappa));
  Solver.solve();
  return;
 }
```

Il metodo `coloring` introduce nello store il vincolo `regioni.eq(colori)` in modo che ad ogni regione venga assegnato un colore, poi tramite il metodo `forall` introduce il vincolo $\{R\}.nin(mappa)$ per ogni regione R appartenente all'insieme `regioni`.

Di seguito riportiamo un main di prova per la classe `Coloring`. Nell'esempio che consideriamo l'insieme `regioni` è dato da {`R1,R2,R3`}, l'insieme `mappa` è dato da {{`R1,R2`},{`R2,R3`}}, l'insieme `colori` è dato da {`rosso,blu`}.

```
 public static void main (String[] args)
 throws IOException,Fallimento
 {
  Lvar R1 = new Lvar();
  Lvar R2 = new Lvar();
  Lvar R3 = new Lvar();
  Lvar[] regioni = {R1,R2,R3};
  String[] colori = {"rosso","blu"};
  Lvar[] n1 = {R1,R2};
  Lvar[] n2 = {R2,R3};
  Set m1 = new Set(n1);
  Set m2 = new Set(n2);
  Set Regions = new Set(regioni);              // Regions = {R1,R2,R3}
  Set Colors = new Set(colori);                // colors = {rosso,blu}
  Set Map = new Set(Set.vuoto.ins(m2).ins(m1));// Map = {{R1,R2},{R2,R3}}

  coloring(Regions,Map,Colors);

  System.out.print("\nR1 = ");Lvar.output(R1);
  System.out.print("\nR2 = ");Lvar.output(R2);
  System.out.print("\nR3 = ");Lvar.output(R3);
 }
}
```

Se il problema è consistente, come è in questo caso, il metodo `coloring` determina una possibile soluzione.

Il metodo `coloring` in questo caso introduce i vincoli:

$\{R1, R2, R3\} = \{rosso, blu\}$

$\forall Y \in \{R1, R2, R3\}, (\{Y\} \notin \{\{R1, R2\}, \{R2, R3\}\})$

Il primo vincolo permette di assegnare un colore a ciascuna delle regioni, gli altri, introdotti dal `forall`, sono necessari per determinare se regioni confinanti hanno colori diversi. Infatti i vincoli introdotti dal `forall` sono:

$\{R1\} \notin \{\{R1, R2\}, \{R2, R3\}\}$

$\{R2\} \notin \{\{R1, R2\}, \{R2, R3\}\}$

$$\{R3\} \notin \{\{R1, R2\}, \{R2, R3\}\}$$

che sono equivalenti ai vincoli:

$$\{R1\} \neq \{R1, R2\}$$

$$\{R1\} \neq \{R2, R3\}$$

$$\{R2\} \neq \{R1, R2\}$$

$\ldots$

Nel nostro caso una possibile soluzione è la seguente:

```
R1 = rosso
R2 = blu
R3 = rosso
```

Capitolo 7

Confronto tra Java e C++

7.1 Introduzione

Nel capitolo 2 abbiamo descritto alcune proposte per la programmazione con vincoli in Java e C++. In questo capitolo descriveremo le principali differenze tra questi due linguaggi e faremo alcune considerazioni su come le peculiarità di Java hanno influito nella realizzazione di JavaSet.

E' idea diffusa che Java come linguaggio Object Oriented sia più semplice del C++. In effetti Java è nato come una semplificazione ed al contempo un miglioramento di C++, quindi i progettisti di Java hanno tentato di affrontare, con l'esperienza maturata negli anni, i problemi tipici affrontati dagli sviluppatori di software. In Java tutte le istruzioni eseguibili devono essere incapsulate all'interno di classi, non sono ammesse funzioni o variabili esterne, quindi Java è un linguaggio che costringe ad immergersi nella programmazione orientata agli oggetti. C++, al contrario permette anche di scrivere codice alla C.

Java supporta tutti i concetti più importanti della programmazione orientata agli oggetti, come information hiding ("mascheramento" dei componenti interni di un oggetto per ottenere una buona separazione tra interfaccia ed implementazione), ereditarietà (capacità di creare un nuovo tipo di oggetto prendendo come base un tipo pre-esistente), polimorfismo (capacità di definire diversi comportamenti a seconda del tipo di ciascun oggetto).

A prima vista C++ e Java appaiono molto simili, ma in effetti presentano anche differenze significative.

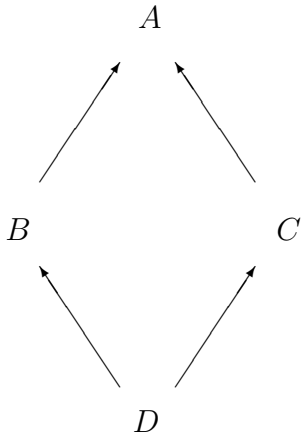
Riassumiamo le principali differenze tra Java e C++ [6]:

- Java non permette l'ereditarietà multipla.
- Java utilizza gli oggetti `Object` al posto delle classi `template`.
- Java utilizza i riferimenti al posto dei puntatori.
- Java utilizza automaticamente la *garbage collection* invece dell'operatore `delete`.
- Java non supporta l'*overloading* degli operatori.
- Java supporta il *multithreading*.

7.2 Ereditarietà singola e multipla

Il C++ impiega l'ereditarietà multipla, grazie alla quale una nuova classe può essere provvista di due o più superclassi.

L'ereditarietà multipla è utile nei casi in cui una nuova classe voglia combinare ed ereditare le caratteristiche di più di una classe. Ma quando c'è più di una superclasse, nei casi in cui il comportamento di una superclasse debba essere ereditato in due modi diversi, insorgono problemi. In C++ questa situazione si presenta nel caso della così detta *ereditarietà a diamante* (cfr. [1]):



Il diagramma mostra quattro classi: A, B, C e D. Le classi B e C ereditano entrambe da A, D eredita sia da B che da C. Il problema sussiste per quanto riguarda l'ereditarietà dell'implementazione nel caso in cui l'implementazione di A memorizzi delle informazioni sullo stato.

Se la classe A avesse ad esempio un attributo pubblico chiamato `a` e se si avesse un oggetto di tipo D chiamato `dref`, non sarebbe semplice capire a cosa dovrebbe riferirsi `dref.a`; infatti potrebbe riferirsi alla copia di `a` presente in B o a quella presente in C, oppure B e C potrebbero condividere un'unica copia di `a` poiché D è anche una A pur essendo contemporaneamente sia una B che una C. La risoluzione di questo problema non è banale.

In Java, per eleminare questi problemi, viene proibita qualsiasi forma di ereditarietà multipla. In JavaSet, ad esempio, non è stato possibile definire la classe `Lvar` come classe derivata sia della classe `Lst` che della classe `Set`. D'altra parte però l'ereditarietà multipla certamente aiuta a modellare meglio il mondo reale. Pensiamo per esempio alla definizione di un metodo per stampare un oggetto: in C++ possiamo dichiarare una classe base `Printable` come segue:

```
class Printable {
   virtual void print();
}
```

Una qualsiasi classe `D` derivata da una classe base `B` può ereditare anche dalla classe `Printable`, basta aggiungere `Printable` all'elenco delle classi base da cui `D` è derivata :

```
class D : public B , virtual public Printable {
   ...
}
```

mentre non è possibile in Java.

Fortunatamente però Java offre un'alternativa: le interfacce (cfr. [1] cap. 4).

Le interfacce permettono di definire dei tipi in modo astratto, infatti non sono dotate di implementazione e non è possibile creare delle loro istanze. Una classe può implementare anche più di una interfaccia e può implementare i metodi di ciascuna interfaccia in qualsiasi modo. Quindi un'interfaccia ha molte più implementazioni di quante ne abbia una classe.

Questo tipo di progettazione porta ad un maggior lavoro per il programmatore, ma permette la flessibilità tipica dell'ereditarietà multipla, evitando però di incorrere nei problemi che insorgerebbero se si ereditassero le implementazioni.

Tornando all'esempio precedente, potremmo considerare la proprietà di essere stampabile come una proprietà generica. Possiamo cioè definire l'interfaccia `Printable` come segue:

```
interface Printable {
   void print()
}
```

`Printable` si limita a dichiarare il metodo `print()`, ma non specifica alcun codice per esso.

La classe `D` può essere definita nel modo seguente:

```
class D extends B implements Printable {
   void print() {...} // implementazione di print()
}
```

Si noti che per segnalare che `D` implementa tutti i metodi di `Printable` si usa la clausola `implements`. E' possibile eseguire il metodo `print()` per qualsiasi classe che implementi `Printable`, ma tutte queste classi non necessitano di essere imparentate tra di loro.

Questo meccanismo può apparire molto simile a definire una classe astratta `Printable` in C++, ma la realtà è molto diversa (anche se in alcune circostanze con ereditarietà multipla di classi astratte si può fare effettivamente qualcosa di simile).

La differenza fondamentale è che `Printable` proprio non esiste come oggetto: non possiamo definire variabili membro né specificare codice per i suoi metodi. Stiamo semplicemente dicendo che `D` possiede un metodo con lo stesso nome di qualunque altra classe che implementi `Printable`, poi ci penserà Java, durante l'esecuzione del programma, a cercare l'indirizzo del metodo giusto, volta per volta. La differenza più profonda è che le classi che implementano le interfacce devono esplicitamente definire il codice per i metodi implementati.

7.3 Classe Object e classi template

Le classi `template` del linguaggio C++ offrono la possibilità, per altro estremamente utile, di definire delle classi che operano su dei dati senza specificare il tipo dei dati quando si definisce la classe.

Java non dispone delle classi `template`(si veda [12]). Si potrebbe pensare che classi simili alle classi `template` del C++ si possono realizzare grazie alla classe `Object` di Java. In effetti in parte è così. La classe `Object` è la radice di qualsiasi altra classe Java sia essa predefinita o definita dall'utente, quindi un oggetto di tipo `Object` può riferirsi ad un istanza di qualunque classe. Questo offre diversi vantaggi, infatti quando all'interno di una classe un campo o un parametro di un metodo è dichiarato di tipo `Object` questo in realtà può essere di qualunque tipo; inoltre grazie alla classe `Object` è possibile definire strutture dati contenenti oggetti di tipi diversi; ad esempio si possono creare array di `Object` e quindi in sostanza array di oggetti qualunque. Gli oggetti di tipo `Object`, quindi si prestano ad essere utilizzati come oggetti generici. Questo significa che in Java c'è meno controllo sui tipi rispetto al C++. Infatti nelle classi template, una volta dichiarato, il parametro di tipo agisce come uno specificatore di tipo all'interno della definizione del template di classe e quindi viene usato esattamente come accade per un tipo predefinito o definito dall'utente in una classe non template. In C++, ad esempio, non è possibile definire un array di oggetti qualunque, mentre in Java basta dichirarlo di tipo `Object`.

Qualche difficoltà però nasce quando non si opera solo con oggetti, ma anche con

i tipi primitivi. Infatti una variabile di tipo primitivo non può essere memorizzata in un `Object`. Quindi non è del tutto vero che la classe `Object` sopperisce alla mancanza delle classi `template` in Java. Infatti esistono estensioni del linguaggio Java, come ad esempio il linguaggio Pizza [13], in cui è previsto il polimorfismo parametrico.

In JavaSet l'esistenza della classe `Object` ci ha offerto la possibilità di definire la classe `Lvar` in modo che il valore di una sua istanza possa essere di diversi tipi. In questo modo non deve essere noto a priori il tipo del valore che potrà assumere la variabile. Al contrario, la libreria C++ ILOG SOLVER, prevede varie classi di variabili. Infatti ogni tipo di variabile è implementato da una classe diversa. Ad esempio, la classe per le variabili intere (`CtIntVar`) è diversa dalla classe delle variabili di tipo reale (`CtFloatVar`) (cfr. sezione 2.3.1 pag.22). Inoltre, grazie alle possibilità offerte dalla classe `Object` in JavaSet gli elementi di un oggetto di tipo `Set` o `Lst` possono essere di tipo diverso.

7.4 Java non ha aritmetica dei puntatori

In C++ esistono ben tre modi diversi per riferirsi ad un oggetto: per valore, per puntatore e per reference. In Java ne esiste uno solo: per reference (che ha un significato diverso che in C++): il reference è una via di mezzo tra il puntatore (richiede allocazione esplicita con `new`) ed il modo per valore (si usa il punto "." e non la freccia "$->$") [1].

Da un punto di vista implementativo certamente il reference è di fatto un puntatore, perché contiene l'indirizzo dell'oggetto a cui "punta"; ma, a differenza dei puntatori tradizionali, non è possibile conoscere tale indirizzo. In questo modo è completamente abolita qualsiasi forma di aritmetica dei puntatori, una delle caratteristiche più "pericolose" del C++.

Praticamente in C++ si può fare di tutto con un puntatore, e questo può portare a codice molto "potente" ma complesso e fonte inesauribile di errori. E' questo il motivo per cui in Java non c'è aritmetica dei puntatori: i puntatori non si possono som-

mare e sottrarre come in C++, però esistono. Quando si crea un oggetto, non si fa altro che assegnare un puntatore ad una zona di memoria in cui ci sono i dati "fisici", gli attributi dell'oggetto stesso. L'operazione di referenziazione/dereferenziazione è automatica. Per accedere ai membri dell'oggetto si usa l'operatore '.' senza dover effettuare nessuna operazione di dereferenziazione.

La situazione è quindi più semplice che in C++, ma comunque bisogna far fronte all'"aliasing". Infatti, passando un oggetto come parametro di un metodo, si passa una copia locale dell'handle (il puntatore) e non dell'oggetto, dunque si crea una situazione "pericolosa". Se il metodo modificherà qualche attributo dell'oggetto tramite il suo handle locale, modificherà in effetti l'unico oggetto esistente, ed all'uscita dal metodo vedremo ovviamente gli attributi modificati, anche se stiamo accedendo all'oggetto tramite un altro handle. Oltretutto in Java non esiste la possibilità di passare un parametro "costante", forzando il compilatore a controllare che non venga modificato. In realtà bisogna pensare ad un handle come ad un puntatore: l'unico modo per garantire che un metodo non modifichi nessun attributo del parametro passato è quello di passare un clone dell'oggetto, in modo che a tutti gli effetti esistano due oggetti diversi. Questo però può peggiorare le prestazioni, ed oltretutto bisogna fare un "deep-copy" dell'oggetto, cioè copiare anche tutti i sottooggetti "contenuti" tramite handle. Un'altra possibile soluzione è quella di utilizzare un oggetto "Proxy" del parametro, che si clona solo se effettivamente necessario, cioè se in effetti si accede in scrittura un attributo dell'oggetto.

7.5 Java ha la garbage collection

Un altro meccanismo relativo alla gestione degli oggetti che Java ci mette a disposizione è la *garbage collection* (si veda [1] cap. 12). In pratica, il runtime di Java è in grado di rendersi conto automaticamente quando un oggetto non serve più ed in tal caso provvede a distruggerlo. Per questo motivo in Java non esistono distruttori né operatore `delete`, ma possiamo dire che in un certo senso è come se essi venissero generati ed usati automaticamente senza che il programmatore se ne renda conto.

Come funziona la garbage collection? Ci sono molti diversi algoritmi per questo meccanismo, ma in generale possiamo dire che il fondamento di tutto è il seguente approccio, detto "mark and release": ad intervalli prefissati di tempo viene eseguita dal codice che esamina prima di tutto le variabili statiche del programma, cercando eventuali puntatori ad oggetti. Ogni volta che si trova un oggetto si marca (mark) come allocata la memoria che usa, poi si procede ricorsivamente ispezionando tutti i puntatori che l'oggetto stesso contiene. Alla fine di questa fase si è giunti per forza a marcare tutti e soli i blocchi di memoria allocati ancora usati; siccome tutti gli altri non sono referenziati da nessuno, per definizione non servono più a niente. Quindi si passa alla seconda fase, il rilascio effettivo (release) dei blocchi di memoria non referenziati.

In Java tutto questo lavoro viene svolto da un processo in background che viene "risvegliato" ad intervalli regolari di tempo, oppure quando c'è necessità di liberare la memoria allocata. I vantaggi della garbage collection sono numerosi e evidenti: non essendo più necessario occuparsi della gestione della memoria ci si può completamente dedicare al problema da risolvere. La gestione della memoria spesso è una responsabilità molto complessa: ci si deve affidare a schemi di progettazione in cui volta per volta deve essere sempre chiaro chi è che "possiede" gli oggetti e quindi ha il "diritto/dovere" di distruggerli.

Gli svantaggi della garbage collection sono genericamente una riduzione della performance: parte della potenza di calcolo viene distolta dall'elaborazione vera e propria, a volte possono verificarsi brevi interruzioni dei programmi quando il garbage collector entra in azione e possono verificarsi dei problemi nel rilascio di risorse. Negli ultimi anni però sono stati sviluppati algoritmi molto più raffinati che riducono sensibilmente questo problema.

La garbage collection, inoltre non costituisce una garanzia del fatto che la memoria sia sempre disponibile per allocare nuovi oggetti: ad esempio si potrebbero creare gli oggetti indefinitamente, situarli all'interno di liste e continuare a fare la stessa cosa finché non vi è più spazio e nessun oggetto non referenziato da reclamare; op-

pure si potrebbero creare specchi di memoria, permettendo che esistano elenchi di oggetti che si riferiscono a oggetti non più necessari . Quindi anche la gestione della memoria richiede comunque un certo grado di attenzione, altrimenti si potrebbe incappare in errori o malfunzionamenti.

7.6 In Java non c'è overloading degli operatori

In Java non è semplicemente possibile effettuare l'overloading degli operatori. Secondo i creatori del linguaggio, questo è un costrutto che può rendere i programmi poco chiari.

Per chiarire questo concetto consideriamo il seguente esempio in C++:

```
class BrokenComplex {
   private: float re, im;
   public: inline float abs() {
           return sqrt(re * re + im * im);
   }
   inline int operator < (BrokenComplex c) {
           return abs() < c.abs();
   }
   inline void operator = (BrokenComplex c) {
           re = c.re; im = c.im;
   }
}
```

Sebbene non ci siano errori di sintassi e tutto il codice possa essere compilato correttamente, la classe BrokenComplex è stata mal progettata. L'operatore "minore di" è stato implementato magari perché il programmatore ha necessità di memorizzare in modo univoco una serie di numeri in una lista, ma è concettualmente sbagliato: i numeri complessi non hanno relazione d'ordine.

Anche l'operatore di assegnamento contiene un errore più sottile: è vero che è in grado di ricopiare un numero complesso in un altro, ma la sua dichiarazione non rispetta la semantica degli operatori di assegnazione del C++. Per definizione gli

operatori di assegnazione in C/C++ ritornano un valore, che è quello dell'oggetto assegnato. In caso contrario non sarà possibile effettuare assegnazioni multiple del tipo `a = b = c`. Quindi il C++ consente di ridefinire gli operatori senza verificare che ne venga rispettata la semantica corretta, cosa che in effetti è praticamente impossibile fare automaticamente.

In Java al posto degli operatori si usano metodi dal nome alfanumerico. Per esempio, per verificare l'uguaglianza tra due oggetti `a` e `b` anziché scrivere `a == b` si usa `a.equals(b)`. Inoltre va osservato che di fatto `a == b` è corretto in Java, ma nel caso in cui `a` e `b` siano oggetti equivale a verificare se due differenti reference puntano allo stesso oggetto, e solo nel caso in cui `a` e `b` siano di tipo primitivo verifica se sono uguali. La stessa sintassi non può quindi essere usata per un confronto tra oggetti differenti, perché le due operazioni sono ben diverse. In C++ non c'è questo problema: se `a` e `b` fossero due oggetti specificati by value, cioè dichiarati come `MyObject a, b` allora chiaramente `a == b` sarebbe un confronto tra oggetti diversi. Se fossero invece puntatori `MyObject *a, *b;` allora `a == b` sarebbe il confronto tra riferimenti allo stesso oggetto, mentre il confronto tra oggetti diversi sarebbe `*a == *b`. Tutto questo perché in C++ ci sono diversi modi per riferirsi ad un oggetto. Per questi motivi, quando leggiamo in un programma C++ `a == b`, siamo obbligati ad andare a vedere come sono stati dichiarati `a` e `b` per capire di che operazione si tratta. Java, in omaggio ai criteri di semplicità e chiarezza, elimina queste ambiguità. D'altraparte però l'overloding degli oreratori è una possibilità molto utile del C++ di cui spesso si sente la mancanza quando si programma in Java.

In JavaSet ad esempio, come si è già detto nella sezione 5.4.3, la mancanza dell'overloading degli operatori ci ha precluso la possibilità di realizzare molte operazioni in modo più naturale ed espressivo utilizzando gli operatori al posto dei metodi alfanumerici.

7.7 Il multithreading

Il multithreading è una forma di programmazione concorrente per cui esistono più "flussi di esecuzione" (threads) attivi contemporaneamente condividendo gli spazi di indirizzamento, cioè di fatto avendo accesso alle stesse istanze di variabili.

Un'alternativa al multithreading è il multitasking, nel quale i "flussi di esecuzione" (chiamati in questo caso task o processi) hanno un loro proprio spazio degli indirizzi, pertanto tutte le istanze di variabili sono duplicate. Il C nacque quando il multithreading di fatto non era ancora stato proposto, pertanto le risorse erano non condivise per default. Si potevano creare risorse condivise (per esempio memoria), ma bisognava farlo esplicitamente e pertanto era chiaro che il programmatore dovesse prendersene cura in maniera opportuna. Con il multithreading il C evidenzia i suoi limiti. A parte il fatto che ogni variabile condivisa richiede di essere protetta esplicitamente, una buona parte delle funzioni di libreria standard risulta inutilizzabile in quanto non solo la loro implementazione, ma anche la semantica, fa uso di variabili statiche per restituire i risultati. Più thread paralleli possono causare conflitti usando contemporaneamente la stessa variabile statica e non è possibile implementare in modo trasparente un metodo di sincronizzazione perché la variabile statica è di fatto accessibile dall'esterno.

Il C++ non può modificare questa situazione a causa dei vincoli i compatibilità che lo legano al suo predecessore e si limita a permetterci di definire librerie di classi (per le quali peraltro non esiste alcuno standard universalmente riconosciuto) per facilitare il controllo dei thread.

Il modello di multithreading di Java è invece elegante e compatto. Interi metodi o solo sezioni specifiche possono essere semplicemente definite come synchronized e la gestione della sincronizzazione è trasparente. Tutti i package standard di Java ovviamente sono stati progettati coerentemente e possono essere usati senza alcun problema. La facilità con cui è possibile creare e controllare un thread parallelo in Java ne rendono possibile l'utilizzo senza doversi porre troppi problemi, consentendo di utilizzare modelli di programmazione più potenti.

Capitolo 8

Conclusioni e lavoro futuro

In questa tesi abbiamo presentato una libreria che estende il paradigma orientato agli oggetti del linguaggio Java con la programmazione con vincoli tipica dei linguaggi logici con vincoli. Attraverso le classi Java che costituiscono JavaSet abbiamo cercato di riprodurre alcune caratteristiche fondamentali dei linguaggi CLP, quali l'utilizzo di variabili logiche, l'unificazione, il non determinismo, la risoluzione dei vincoli.

Gli insiemi rivestono un ruolo fondamentale nella libreria, in quanto sono le uniche strutture dati su cui sono definite operazioni non deterministiche. Infatti l'unificazione e le altre operazioni sugli insiemi sono inerentemente e naturalmente non deterministiche.

La realizzazione del non determinismo non è stata immediata. In JavaSet il non determinismo è stato realizzato utilizzando i punti di scelta e il backtracking. Tutte le volte che per un vincolo si presentano diverse possibili soluzioni, il risolutore crea un punto di scelta e lo aggiunge allo stack delle alternative, poi procede nella computazione considerando una delle possibili soluzioni per quel vincolo. Se la soluzione considerata causa un'inconsistenza, l'algoritmo applica il backtracking cioè estrae un punto di scelta dalla pila, riporta le variabili e lo store nello stato memorizzato in quel punto di scelta e continua esaminando un'altra alternativa. Se non ci sono più alternative la computazione fallisce.

Nella nostra realizzazione del non determinismo il backtracking agisce sui vincoli e sulle variabili, cioè riporta il constraint store e le variabili (`Lvar`, `Lst`, `Set`) in uno stato precedente, in modo da annullare gli effetti della risoluzione di vincoli che hanno portato ad una inconsistenza. Tuttavia durante il backtracking non vengono rivalutate le eventuali strutture di controllo all'interno delle quali sono stati introdotti i vincoli.

Per chiarire questo aspetto consideriamo, ad esempio, la seguente struttura di controllo `if-else` in cui L è una lista, v_1 e v_2 sono due vincoli:

$Solver.add(L \ in \ \{[\ ], [X]\});$
`if` $(L == [\ ])\ Solver.add(v_1);$
`else` $Solver.add(v_2);$

Se al momento della valutazione dell'istruzione `if`, la lista L è inizializzata con la lista [ ], allora viene aggiunto allo store il vincolo v_1, altrimenti viene aggiunto il vincolo v_2. Supponiamo, ad esempio, che venga aggiunto allo store il vincolo v_1. Se in seguito, in conseguenza del backtracking, alla lista L viene assegnata la lista $[X]$, anziché la lista [ ], si ha che al posto del vincolo v_1 dovrebbe essere valutato il vincolo v_2, invece nello store resta memorizzato v_1, e il vincolo v_2 non viene considerato, in quanto durante il backtracking non viene rivalutata la struttura di controllo `if`.

Lo stesso vale anche per le altre strutture di controllo: `while`, `do-while`, `switch`. Infatti, mentre per la struttura `for` c'è il metodo `forall` che permette di realizzare un ciclo sensibile al backtracking, non esiste nulla di simile per le altre strutture di controllo.

Un approccio alternativo per la realizzazione del non determinismo, che risolverebbe questo problema, potrebbe essere, a grandi linee, il seguente: ciascun programma per la risoluzione di un problema di soddisfacimento di vincoli dovrebbe essere un'istanza di un'apposita classe creata per la risoluzione di quel particolare problema. Questa classe dovrebbe implementare un metodo in grado di eseguire il programma. Un programma dovrebbe essere costituito da un blocco `switch-case`

che permetterebbe di etichettare le istruzioni che compaiono dopo l'introduzione di un punto di scelta. L'esecuzione di un programma avverrebbe eseguendo le istruzioni contenute nei vari blocchi `case` partendo dal primo. Durante la risoluzione dei vincoli, in caso di backtracking, il risolutore procederebbe considerando un punto di scelta che ha ancora delle alternative non esplorate e scegliendo una delle possibili soluzioni. A questo punto il controllo verrebbe trasferito al blocco `case` etichettato con il valore memorizzato nel punto di scelta considerato, in questo modo verrebbero rieseguite le istruzioni successive al punto in cui è stato creato quel punto di scelta.

Quindi un programma che esegue la struttura `if-else` vista in precedenza potrebbe essere realizzato definendo la seguente classe:

```
public class Prova
{
   private static Lst l;
   private static Lvar x;
   ...      // costruttori della classe Prova
   public static void execute()
   {
    switch(control){
      case 0:  Solver.add(l.in{[ ],[x]},1);
      case 1:  if (l == [ ])Solver.add(v1);
               else Solver.add(v2);
    }
    return;
   }
}
```

Il metodo `main` del programma dovrebbe dunque creare un'istanza della classe `Prova` e invocare su di essa il metodo `execute`:

```
Prova p = new Prova(L,X);
p.execute();
```

In caso di backtracking, verrebbe rieseguita la porzione di codice successiva al punto di scelta considerato e si avrebbe quindi il vantaggio di riuscire a riesaminare tutte le istruzioni eseguite e non solo i vincoli, come accade invece in JavaSet. Tuttavia questo approccio costringerebbe ad una modalità di programmazione molto innaturale. Per risolvere questo problema bisognerebbe realizzare un "pre-processor" in modo da permettere all'utente di scrivere i propri programmi in modo più usuale.

L'approccio più naturale per la realizzazione del non determinismo in Java sarebbe stato quello di ricorrere al multithreading, cioè alla possibilità del linguaggio Java di avere più "flussi di esecuzione" attivi contemporaneamente. Tuttavia non è stato possibile realizzare il non determinismo in questo modo. Infatti si potrebbe pensare di risolvere i vincoli non deterministici definendo più thread. Ciascun thread dovrebbe esplorare una delle possibili alternative terminando in caso di fallimento. In questo modo però tutti i thread generati durante la risoluzione di un vincolo, che non giungano ad un fallimento, dovrebbero poi condividere la stessa porzione di codice, cioè ciascun thread dovrebbe eseguire tutte le istruzioni successive al punto in cui il flusso di esecuzione si è diramato. Il multitreading del Java però non supporta un meccanismo di questo tipo.

Il non determinismo, così come è realizzato in JavaSet, ha quindi il vantaggio di permettere una modalità di programmazione abbastanza naturale, ma come si è detto persenta anche dei punti critici. Un progetto per il futuro è quello di realizzare un'introduzione "condizionale" dei vincoli nello store, cioè oltre al metodo `add` per l'introduzione dei vincoli, vorremmo realizzare altri metodi che permettano di introdurre i vincoli nello store in modo però da poter rivalutare, durante il backtracking, le condizioni che hanno portato all'introduzione nello store di quel vincolo ed eventualmente sostituirlo.

Tornando all'esempio precedente, per fare in modo che la sruttura `if-else` considerata venga rivalutata in caso di backtracking si potrebbe procedere nel modo seguente:

$Solver.add(L\ .in\ \{[\,], [X]\});$
$Solver.add\_cond((L\ .eq\ ([\,])\ , v_1, v_2);$

Il metodo `add_cond` dovrà essere definito in modo da creare e aggiungere allo store un nuovo oggetto nel quale siano memorizzatati la condizione da verificare e i vincoli da risolvere nei vari casi. Poiché gli elementi memorizzati nello store sono oggetti di tipo `StoreElem`, anche gli oggetti introdotti dal metodo `add_cond` dovranno essere istanze di questa classe. Alla classe `StoreElem` dovranno quindi essere aggiunti dei campi nei quali memorizzare la condizione da verificare, nel caso in cui ci sia, e i vincoli tra cui scegliere in base alla condizione. Questi campi non saranno utilizzati nel caso di vincoli introdotti tramite il metodo `add`. In questo modo, in caso di backtracking, verrà rivalutata ogni volta la condizione introdotta, per decidere quale vincolo considerare.

Un altro aspetto che vorremmo migliorare in futuro riguarda i costi computazionali della risoluzione dei vincoli. Infatti i vincoli tra insiemi vengono valutati nello stesso modo indipendentemente dal fatto che gli insiemi coinvolti siano completamente noti o contengano variabili non inizializzate oppure che siano limitati o non limitati. Quindi ad esempio il vincolo $\{1, 2\}.subset(\{1, 5, 3, 2\})$ e il vincolo $\{X, Y|R\}.subset(\{V, W, Z|S\})$ con R ed S insiemi non inizializzati e X, Y, V, W e Z variabili qualunque, anche non inizializzate, vengono risolti utilizzando le stesse regole di riscrittura non deterministiche. E' ovvio che se è noto che gli insiemi coinvolti in un vincolo sono completamente noti e limitati, i costi computazionali per la risoluzione di quel vincolo possono essere notevolmente migliorati. Quindi l'idea per il futuro è quella di introdurre dei controlli, prima di iniziare la risoluzione dei vincoli, in modo da stabilire di che tipo sono gli insiemi coinvolti così da poter differenziare la risoluzione.

Un altro aspetto che sarebbe interessante valutare riguarda i vincoli di confronto e i vincoli aritmetici. Infatti questi vincoli in JavaSet possono essere applicati solo a variabili note. Per poterli trattare come gli altri vincoli, cioè per poterli applicare

anche a variabili non inizializzate, sarebbe necessario disporre di un risolutore adatto, in grado di risolvere sistemi di equazioni e disequazioni in più incognite. Sarebbe quindi interessante valutare la possibilità di integrare il risolutore di JavaSet con altri risolutori. Ovviamente però l'integrazione tra risolutori non è un processo banale.

Bibliografia

[1] Ken Arnold, James Gosling, David Holmes. *Java. Manuale ufficiale.* Addison-Wesley, 2001 ISBN 88-7192-097-X.

[2] Krzysztof R. Apt and Andrea Schaerf. *The Alma Project, or How First-Order Logic Can Help Us in Imperative Programming.* 1999 http://www.cwi.nl/∼apt.

[3] Krzysztof R. Apt and Andrea Schaerf. *Integrating Constraints into an Imperative Programming Language.* 1999 http://www.cwi.nl/∼apt.

[4] Marco Bertacca, Andrea Guidi. *Introduzione a Java.* McGraw-Hill, 2000 ISBN 88-386-0849-0.

[5] Andy Chun. *Constraint programming in Java with JSolver.* In Proc. Practical Applications of Constraint Logic Programming PACLP99, 1999. http://www.city.edu.hk/∼eehwchun.

[6] Michael C. Daconta *Java for C/C++ Programmers.* New York: John Wiley and Sons.

[7] D.Diaz, P.Codognet. *A minimal extension of the* Wam *for CLP(fd).* Proceedings of the 10th International Conference on Logic Programming, 1993. http://www.daimi.au.dk/∼beta/Papers/BETAconstraints.

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

[9] P. Van Hentenryck. *Constraints satisfaction in Logic Programming.*
MIT Press, 1989.

[10] ILOG 2001. *ILOG Optimisation Suite White Paper.*
http://www.ilog.com/products/optimisation/tech/optimisation_whitepaper.pdf.

[11] Michel Leconte, Jean-Francois Puget. *Beyond the Glass Box: Constraints as Objects.* Proc. of the 1995 International Symposium on Logic Programming, MIT press, pp. 513-527.
http://www.ilog.com.

[12] Stanley B. Lippman, Josée Lajoie *C++ Corso di programmazione.*
Addison-Wesley, 2000 ISBN 88-7192-071-6.

[13] Martin Odersky, Philip Wadler. *Pizza into Java: Traslating theory into practice.*
Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.

[14] W.J Older, F. Benhamou. *Programming in CLP(BNR).*
Proceeding of PPCP'93, Providence, RI, April 1993.

[15] Jean-Francois Puget. *A C++ Implementation of CLP.*
Ilog Solver Collected Papers, Ilog tech repart. 1994
http://www.ilog.com.

[16] Gianfranco Rossi. *Set-based Nonodeterministic Declarative Programming in* SINGLETON. 11th International Workshop on Functional and (constraint) Logic Programming. Grado(IT), June 20-22, 2002.

[17] Neng-Fa Zhou. *B-Prolog: User's manual, Version 3.1.*
Kyushu Institute of Tecnology, 1998.
http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.htlm.

[18] Neng-Fa Zhou. *Building Java Applets by using DJ - a Java Based Constraint Language.*
http://www.sci.brooklyn.cuny.edu/∼zhou.

[19] Neng-Fa Zhou. *DJ: Declarative Java, Version 0.5, User's manual.*
Kyushu Institute of Tecnology, 1999.
http://www.cad.mse.kyutech.ac.jp/people/zhou/dj.htlm.