UNIVERSITÀ DI PARMA

Dipartimento di Scienze Matematiche, Fisiche ed Informatiche

Corso di Laurea in Informatica

Estensioni ed uso degli Insiemi Intensionali Ristretti in JSetL

Extensions and applications of Restricted Intensional Sets in JSetL

Relatore:

Chiar.mo Prof. Gianfranco Rossi

Tesi di Laurea di:

Andrea Fois

ANNO ACCADEMICO 2017-2018

Alla mia famiglia, alla mia ragazza, ai miei amici.

"Computer science inverts the normal. In normal science, you're given a world, and your job is to find out the rules. In computer science, you give the computer the rules, and it creates the world."

Alan Kay

Indice

Introduzione **1**

0.1 Insiemi intensionali 1

0.1.1 Notazione Z 1

0.2 Programmazione con insiemi intensionali 2

0.3 Insiemi Intensionali Ristretti 3

0.3.1 RIS 4

0.4 Scopo del lavoro 5

1 Insiemi intensionali ristretti in JSetL **6**

1.1 JSetL 6

1.1.1 Strutture dati di JSetL 6

1.1.2 JSetL come constraint solver 9

1.1.3 Un primo programma con JSetL 11

1.1.4 LCollection parzialmente specificate 13

1.2 La classe Ris 14

1.2.1 Constraints sui Ris 15

2 Usi dei RIS **17**

2.1 Usare i RIS come funzioni 17

2.1.1 Passaggio di funzioni in Java 17

2.1.2 Passaggio di funzioni coi RIS 19

2.1.3 Passaggio di funzioni con Lambda Expressions 20

2.1.4 Filtrare un array in Java 21

2.2 Usare i Ris come relazioni 22

2.2.1 Un esempio: il dizionario Italiano-Inglese 22

3 Nuovi vincoli per i Ris **31**

3.1 Vincolo di disgiunzione 31

3.1.1 Uso del disgiunto nei Ris di JSetL 32

3.2 Vincolo di unione 34

3.2.1 Uso dell'unione nei Ris di JSetL 34

3.3 Vincoli derivati 36

4 Riscrittura dei vincoli di inequalità **37**

4.1 Problemi coi vincoli su LSet 37

4.2 Problemi coi vincoli sui Ris 38

4.3 Soluzione adottata 39

5 Un'estensione ai RIS: variabili esistenziali nel filtro e nel pattern **40**

5.1 Le istanze del control term 40

5.2 Variabili esistenziali aggiuntive 41

5.3 Un programma di esempio 42

6 Conclusioni e sviluppi futuri **44**

Riferimenti bibliografici **46**

Introduzione

0.1 Insiemi intensionali

Gli insiemi intensionali sono insiemi dati descrivendo le proprietà delle entità che contengono; si distinguono dunque dagli insiemi dati per enumerazione che, invece, elencano tutti gli oggetti contenuti.

Esempio 0.1.1.

$$\{n \in \mathbb{N} \mid n > 0 \land n < 10 \land n \bmod 2 \cong 0\} = \{2, 4, 6, 8\}.$$

0.1.1 Notazione Z

La notazione Z(7)(dal nome di Zermelo delle teoria degli insiemi di Zermelo-Fraenkel) è un linguaggio di specifica formale basata sulla notazione usata nella teoria assiomatica degli insiemi e nella logica del primo ordine. La notazione Z prevede un modo semplice di esprimere insiemi intensionali.

$$S = \{x : D \mid F(x) \bullet P(x)\}$$

dove:

- x è la variabile di controllo,
- D è un insieme detto dominio,
- $F(x)$ è il filtro ed è un predicato che la variabile di controllo deve verificare,

- $P(x)$ è il pattern ed è una espressione contenente il termine di controllo e il suo risultato è il valore effettivamente collezionato dall'insieme.

La seguente equazione insiemistica mostra lo stesso insieme definito in notazione Z e con la notazione usuale degli insiemi intensionali.

Esempio 0.1.2.

$$\{n : \mathbb{N} \mid n > 0 \land n < 10 \land n \bmod 2 \cong 0 \bullet 2n\} = \{2n \in \mathbb{N} \mid n > 0 \land n < 10 \land n \bmod 2 \cong 0\}.$$

0.2 Programmazione con insiemi intensionali

Gli insiemi intensionali permettono di esprimere in maniera elegante e concisa collezioni di elementi aventi certe proprietà senza bisogno di elencarli. Alcuni linguaggi permettono l'uso di un certo tipo di insiemi intensionali. In Javascript si può ottenere qualcosa di simile ad un insieme intensionale attraverso l'uso delle funzioni generatrici(9), funzioni che ritornano generatori, oggetti usati in Javascript in maniera analoga agli iteratori di Java e c++. La seguente funzione generatrice restituisce un generatore che permette di generare i numeri della sequenza di fibonacci.

```
function* fibonacci(){
   var a = 0;
   var b = 1;
   while(true){
          yield a;
      a = a + b;
      yield b;
      b = a + b;
   }
}
```

Il seguente stralcio di codice stampa sulla console i primi 20 numeri di fibonacci sfruttando la funzione generatrice definita sopra.

```
var fib = fibonacci();
for (let i = 0; i < 20; ++i)
```

```
console.log(fib.next().value);
```

Un altro tipo di approccio è quello usato da minizinc, che permette di definire insiemi con dominio specificato come variabili di decisione, andando a verificare quali elementi del dominio appartengono all'insieme provandoli tutti.
I limiti degli approcci descritti sono evidenti: non è posibile lavorare con insiemi parzialmente specificati e per poter lavorare in qualche modo con l'insieme è sempre necessario enumerare i suoi elementi. Tale operazione non è sempre possibile in quanto, ad esempio, il dominio dell'insieme potrebbe non essere noto. Gli approcci di cui sopra si basano sul fatto che il dominio dell'insieme su cui si lavora sia noto: gli interi rappresentabili correttamente dai numeri in virgola mobile a precisione doppia nel caso di Javascript(dominio implicito nella definizione della funzione) e il dominio che è necessario specificare in un programma minizinc quando si definiscono i set senza enumerare gli elementi. Non è dunque possibile risolvere il seguente vincolo:

$$16 \in \{x : D \mid x \bmod 2 = 0 \bullet 2x\}$$

Se D è una variabile libera. La funzione generatrice data con Javascript ha inoltre un altro problema: non termina mai e pertanto non è possibile verificare la non appartenenza di un intero all'insieme dei numeri di Fibonacci.

0.3 Insiemi Intensionali Ristretti

Per risolvere i problemi evidenziati è necessario un approccio diverso: da un lato bisogna restringere il potere espressivo degli insiemi intensionali, anche oltre quello fatto normalmente per evitare il paradosso di Russel, limitandoci a insiemi intensionali con un dominio finito (ma non necessariamente specificato), dall'altro bisogna trattare le operazioni sugli insiemi intensionali come vincoli, ad esempio verificando l'appartenenza in un elemento all'insieme verificando che esso soddisfi il filtro, senza bisogno di generare l'insieme

estensionale corrispondente all'insieme intensionale. In (4) è definito formalmente un linguaggio in grado di specificare insiemi intensionali ristretti e specificare 4 vincoli su di essi($\in, \notin, =, \neq$). È inoltre specificata una procedura di decisione per tali vincoli.

0.3.1 RIS

Gli Insiemi Intensionali Ristretti(d'ora in poi RIS, da *Restricted Intensional Sets*) sono insiemi intensionali con alcune restrizioni introdotti in (4). Ne vediamo prima la sintassi(Z-like) e poi la semantica. Un RIS è un oggetto della forma

$$\{x : D \mid F(x) \bullet P(x)\}$$

dove:

- x è una variabile libera,
- D è un insieme detto dominio,
- $F(x)$ è il filtro ed è un vincolo,
- $P(x)$ è il pattern ed è una espressione contenente x.

Da notare è che in (4) il *control term* è limitato ad essere una variabile o una coppia di variabili mentre il *pattern* può essere solo della forma x, (x, t) o (t, x) con x *control term* e t un qualsasi termine del linguaggio sottostante.

Esempio 0.3.1.

$$\{n : \mathbb{N} \mid n > 0 \land n < 10 \land n \bmod 2 \cong 0 \bullet 2 \cdot n\}$$

Bisogna notare anche che il filtro ed il pattern possono contenere altre variabili oltre al *control term*, libere e non.

Esempio 0.3.2.

$$\{n : \mathbb{N} \mid n > 0 \land n < 10 \land n \bmod 2 \cong y \bullet w \cdot n\}$$

Per convenzione omettiamo il filtro se esso è *true*, inoltre omettiamo il *pattern* se è uguale al *control term*.

Esempio 0.3.3.

$$\{x : \{1, 5, "a"\} \,|\, true \bullet [x, x]\} \equiv \{x : \{1, 5, "a"\} \bullet [x, x]\}$$

Esempio 0.3.4.

$$\{x : \{1, 5, 9\} \,|\, x < 3 \bullet x\} \equiv \{x : \{1, 5, "a"\} \,|\, x < 3\}$$

La semantica informale della scrittura $\{x : D \,|\, F(x) \bullet P(x)\}$ è la seguente: l'insieme dei $P(x)$ con $x \in D \wedge F(x)$. Risulta evidente che se i domini del *control term* e di tutte le variabili esistenziali del RIS sono finiti allora anche il RIS è un insieme finito.

0.4 Scopo del lavoro

Lo scopo del lavoro è stato quello di utilizzare l'implementazione dei Ris in JSetL al fine di verificare l'utilità dell'approccio proposto per gli insiemi intensionali all'interno di un linguaggio di programmazione convenzionale come Java, dal punto di vista della semplicità di scrittura, comprensione e manutenzione del codice. Viene inoltre estesa ed arricchita la libreria JSetL con nuovi vincoli sui Ris e col supporto a Ris non previsti dal linguaggio originariamente descritto in (4). Nel primo capitolo verrà introdotta la libreria JSetL introdotta in (3) e la prima implementazione dei Ris descritta in (6). Il secondo capitolo si concentra invece sugli usi dei Ris, in particolare per passare funzioni e relazioni (in senso matematico) a metodi Java. Nel terzo capitolo vengono descritti due nuovi vincoli sui Ris, l'unione e la disgiunzione, e ne viene spiegata l'implementazione. Nel quarto capitolo si risolve un problema che era presenta nella procedura di decisione implementata in JSetL e che riguardava anche i Ris. Nel quinto capitolo si descrive un'estensione realizzata alla procedura di risoluzione dei vincoli che permette di risolvere correttamente vincoli su Ris con variabili esistenziali nel proprio filtro e pattern, sotto certe pattuizioni.

Capitolo 1

Insiemi intensionali ristretti in JSetL

1.1 JSetL

JSetL è una libreria Java mirata alla programmazione logica con vincoli basata su insiemi. JSetL è un'implementazione $CLP(SET)$.

1.1.1 Strutture dati di JSetL

JSetL mette a disposizione del programmatore alcuni tipi di dato tipici della programmazione logica come le variabili logiche, le liste e gli insiemi. Ora descriviamo brevemente quali sono le principali strutture dati fornite e da quali classi Java sono implementate.

- Variabili logiche che possono contenere qualunque oggetto Java, ad esempio un `Integer` o una `String`. La classe `LVar` le implementa. Possiamo creare variabili logiche con un valore assegnato (inizializzate) o senza un valore assegnato (non inizializzate), con un nome assegnato o senza un nome assegnato.

```
LVar x1 = new LVar(); // crea una variabile logica non inizializzata
    e non gli assegna un nome
```

```
LVar x2 = new LVar(''X''); // crea la variabile logica non
    inizializzata di nome X
LVar x3 = new LVar(12345); // crea una variabile logica con valore
    12345
```

- Variabili logiche intere, simili al punto precedente ma contengono solo valori interi e pertanto permettono di esprimere vincoli numerici. Sono implementate da `IntLVar`, sottoclasse di `LVar`.

```
IntLVar y1 = new IntLVar(); // crea una variabile logica intera
    libera senza nome
IntLVar y2 = new IntLVar(6789); // crea la variabile logica intera
    di valore 6789
IntLVar y3 = new IntLVar(''Y'', 1111); // crea la variabile logica
    Y con valore 1111
```

- Insiemi dati per enumerazione, che contengono elementi anche non omogenei in tipo. Sono implementati dalla classe `LSet`. È possibile definire un insieme vuoto attraverso la seguente chiamata

```
LSet emptySet = LSet.empty();
```

È possibile creare insiemi completamente variabili con nome e senza nome.

```
LSet set = new LSet(); // nuovo insieme variabile senza nome
LSet set = new LSet(''S''); // nuovo insieme variabile di nome S
```

È anche possibile assegnare nomi esterni a qualisasi oggetto logico, dopo aver creato l'oggetto, con il metodo `setName()`.

```
LSet set = new LSet();
set.setName(''S'');
```

Dato un insieme A è possibile creare un insieme B che è pari ad A unito con un certo singoletto tramite il metodo `ins` di `LSet`.

```
LSet a = LSet.empty().ins(2); // a ={2}
LSet b = a.ins(3); // b = {2,3}
```

Creando innanzitutto un insieme variabile e poi inserendo degli elementi è possibile creare insiemi parzialmente specificati, ovvero formati da una sequenza di elementi noti seguita da un resto variabile.

```
LSet d = new LSet(''C'').ins(2).ins(3); // d = {2,3|C}, i.e. {2,3}
    unito C
```

- Insiemi simili a quelli del punto precedente ma che possono contenere solo valori interi, e pertanto possono fungere da dominio di `IntLVar`. Essi sono implementati da `IntLSet` che è sottoclasse di `LSet`. Possono essere dichiarati nello stesso modo degli `LSet`.

```
IntLSet emptyIntLSet = IntLSet.empty(); // {}
IntLSet a = new IntLSet(); // set di interi completamente variabile
IntLSet b = new IntLSet(''B''); // set di interi variabile di nome B
```

Anche l'aggiunta di elementi è analoga agli `LSet`.

```
IntLSet a = IntLSet.empty().ins(3).ins(2); // a = {2, 3}
IntLSet b = new IntLSet(''R'').ins(9).ins(8); // b = {9, 8 | R}
```

- Liste di elementi, anche ripetuti e possibilmente disomogenei nel tipo. Esse sono implementate da `LList`. Anche per le liste le dichiarazioni e l'inserimento sono analoghi rispetto agli `LSet` e si applicano gli stessi ragionamenti per le liste parzialmente specificate.

```
LList list = new LList(); // lista completamente variabile
LList bb = new LList(''R'').ins(3).ins(3); // lista con parte non
    specificata: [3, 3 | R]
LList emptyLList = LList.empty(); // []
LList cc = LList.empty().ins(1).ins(1); // [1, 1]
```

Poichè l'ordine è importante nelle liste bisogna notare che la lista risultante da inserimenti concatenati con `ins()` ha gli elementi nell'ordine inverso rispetto a come appaiono nel sorgente. Per questo motivo esiste il metodo statico `LList.mkPair(a,b)`.

```
LList aa = LList.empty().ins(2).ins(1); // [1, 2]
LList bb = LList.mkPair(1,2); // [1, 2]
```

- Insiemi intensionali ristretti, l'oggetto di questa tesi, implementati dalla classe `Ris`. Questi oggetti verranno trattati ampiamente nel seguito, per ora è sufficiente dire che `Ris` è una sottoclasse di `LSet`.
- La classe `LCollection`, mai usata direttamente dall'utente funge da classe base per `LSet` e `LList` e ci permetterà di parlare in generale di alcune delle funzionalità comuni a queste classi.

1.1.2 JSetL come constraint solver

Oltre alle strutture dati viste prima JSetL offre numerose funzionalità per esprimere vincoli logici su di esse e per risolverli, trovando le soluzioni e unificando le variabili coi valori corretti, se esistono. In JSetL i vincoli sono istanze della classe Constraint che a sua volta è una lista di vincoli atomici, istanze di `AConstraint`. Passiamo in rassegna alcuni dei principali vincoli che JSetL permette di esprimere.

- L'uguaglianza e la disuguaglianza, `Constraint` creati rispettivamente dai metodi `eq` e `neq` di `LVar` e `LCollection`.

```
Constraint uguaglianza = A.eq(B);
```

- L'appartenenza e la non appartenenza di una `LVar` a un insieme tramite il metodo `in` o `nin` di `LVar`. Dualmente `LSet` prevede il metodo contains che che accetta un `Object` qualsiasi.

```
Constraint c1 = lvar.in(S);
Constraint c2 = S.contains(lvar);
```

- L'unione fra insiemi, ovvero il metodo `union` di `LSet`.

```
A.union(B, C); // A = B ∪ C
```

- La disgiunzione fra insiemi tramite il metodo `disj` di `LSet`.

```
A.disj(B); // A || B
```

JSetL prevede inoltre la possibilità di aggiungere questi vincoli ad un solver in grado di risolverli facendo deduzioni sugli oggetti coinvolti. I vincoli vengono in primo luogo aggiunti al solver e poi vengono risolti. Il solver è in grado di restituire tutte le soluzioni e non solo la prima. Per offrire queste funzionalità il solver utilizza al suo interno un meccanismo di backtracking con punti di scelta per ciascun vincolo. Vediamo ora in dettaglio la classe `SolverClass` che implementa il solver e i suoi metodi principali.

- Costruttore senza parametri.

```
SolverClass solver = new SolverClass();
```

- Metodo `add` che permette di aggiungere vincoli al solver.

```
SolverClass solver = new SolverClass();
solver.add(A.neq(B)); // A != B
```

- Metodo `check()` che restituisce un booleano `true` se il solver è riuscito a trovare una soluzione, `false` altrimenti.

```
SolverClass solver = new SolverClass();
solver.add(IntLSet.empty().contains(1));
solver.check(); // FALSE
```

- Metodo `void solve()`. Cerca di risolvere i vincoli, se non riesce a farlo solleva un'eccezione di tipo `Failure`.

```
SolverClass solver = new SolverClass();
IntLSet A = new IntLSet();
IntLSet B = IntLSet.empty().ins(10);
```

```
solver.add(A.eq(B));

solver.solve(); // OK
System.out.println(A); // {10}
```

```
SolverClass solver = new SolverClass();
IntLSet A = IntLSet.empty().ins(11);
IntLSet B = IntLSet.empty().ins(10);
solver.add(A.eq(B));

solver.solve(); // Solleva eccezione Failure
```

- Metodo `nextSolution()`. Analogo alla `check()` e da usare dopo una sua chiamata. Cerca di trovare un'altra soluzione andando ad esplorare i punti di scelta ancora aperti. Se non trova un'altra soluzione restituisce `false`, altrimenti `true`.

```
SolverClass solver = new SolverClass();
IntLVar x = new IntLVar();
IntLSet S = IntLSet.empty().ins(1).ins(2);
solver.add(x.in(S));
solver.check(); // TRUE, x = 2
solver.nextSolution(); // TRUE, x = 1
solver.nextSolution(); // FALSE
```

1.1.3 Un primo programma con JSetL

Mostriamo ora un esempio di un semplice programma che usa la libreria JSetL. Il programma crea un insieme chiamato set che contiene alcune coppie, dopodiché si crea un solver e si aggiunge il constraint $[x, y] \in set$, si trovano dunque tutte le soluzioni possibili e si stampano a video i valori di `x` e `y` per ciascuna soluzione.

File: PrimoProgrammaJSetL.java

```
package JSetL.tesiFois;
```

```
import JSetL.*;
public class PrimoProgrammaJSetL {
   public static void main(String[] args) throws Failure{
      //set = {[1,10], [2,20], [3,30]}
      LSet set = LSet.empty() // un set vuoto
              .ins(LList.mkPair(1, 10)) //aggiunge [1,10] all'insieme e lo
                   restituisce
              .ins(LList.mkPair(2, 20)) //aggiunge [2,20]
              .ins(LList.mkPair(3, 30)) //aggiunge [3,30]
              .setName("set"); //assegna il nome "set" all'insieme

      LVar x = new LVar("x");
      LVar y = new LVar("y");

      LList pairxy = LList.mkPair(x,y); // [x,y]

      SolverClass solver = new SolverClass();

      //aggiunge al solver la constraint
      //[x,y] in set = {[1,10], [2,20], [3,30]}
      solver.add(set.contains(pairxy));

      solver.solve(); //risolve le constraint e assegna i valori a x e y

      x.output(); //3
      y.output(); //30

      while(solver.nextSolution()) {
         x.output();
         y.output();
      }

      /* output:
          x = 3
         y = 30
         x = 2
         y = 20
         x = 1
```

```
        y = 10
      */
  }

}
```

1.1.4 LCollection parzialmente specificate

Nell'esempio precedente abbiamo visto che le `LCollection` (in particolare le `LList`, nell'esempio) possono contenere anche variabili logiche che non hanno ancora un valore assegnato. Un'altra funzionalità offerta da JSetL è la possibilità di definire `LCollection` parzialmente specificate, e in particolare `LList` e `LSet` parzialmente specificati, ovvero con una lista (possibilmente vuota) di elementi specificati e un resto completamente variabile. Il seguente programma crea un insieme di interi parzialmente specificato, crea un solver e aggiunge il constraint che prevede che un numero che non è ancora (noto essere) nell'insieme appartenga all'insieme e la risolve. Da notare è che dopo aver risolto la constraint l'insieme di partenza è stato effettivamente modificato e ora mostra il nuovo numero fra i suoi elementi, oltre ad avere ancora un resto completamente variabile.

File: JSetLLSetParzialmenteSpecificato.java

```
package JSetL.tesiFois;
import JSetL.*;
public class JSetLLSetParzialmenteSpecificato {
   public static void main(String[] args) throws Failure{
      IntLSet S = new IntLSet() // completamente variabile
               .ins(1)
               .ins(2)
               .ins(3)
               .ins(4)
               .setName("S");
      S.output(); // S = {4,3,2,1|_?}
```

```
      SolverClass solver = new SolverClass();
      solver.add(S.contains(new IntLVar(5)));
      solver.solve();
      S.output(); // S = {4,3,2,1,5|_?}

   }
}
```

1.2 La classe Ris

In JSetL gli insiemi intensionali, in una loro variante ristretta, sono istanze della classe `Ris`. Per creare un `Ris` si usa un costruttore dalla seguente forma.

```
Ris ris = new Ris(controlTerm, dominio, filtro, pattern);
```

Il significato degli argomenti del costruttore è il seguente.

- Il `controlTerm` svolge lo stesso ruolo della variabile di controllo degli insiemi intensionali classici. Può essere una `LVar` o sua sottoclasse o una `LList` o una coppia, implementata in JSetL dalla classe `Pair`.
- Il `dominio` è l'insieme a cui il `controlTerm` deve appartenere, deve dunque essere un `LSet` o sua sottoclasse.
- Il `filtro` è un `Constraint` che il `controlTerm` deve soddisfare per essere usato per generare gli elementi dell'insieme.
- Il `pattern` è una variabile che può contenere il (o essere legata tramite vincoli al) `controlTerm`. Il pattern può essere una singola `LVar` o una `LList` o una `Pair`.

La semantica informale di un Ris come sopra è: l'insieme di tutti i `pattern` tali che il `controlTerm` appartiene al `dominio` e il `filtro` è soddisfatto.

Il seguente esempio mostra come creare un `Ris` che denota l'insieme $\{2x | x \in \{1,2,3\} \wedge x \bmod 2 = 1\}$

```
IntLVar x = new IntLVar();
IntLSet dom = IntLSet.empty().ins(1).ins(2).ins(3);
Ris ris = new Ris(x, dom, x.mod(2).eq(1), x.mul(2));
```

Anche ai `Ris` è possibile assegnare un nome, proprio come per le altre strutture dati di JSetL. In questo caso il primo argomento del costruttore è una `String`.

```
Ris S = Ris(''S'', controlTerm, dominio, filtro, pattern);
```

Esiste (e nei nostri esperimenti è risultato abbastanza comune) il caso in cui il control term sia uguale al pattern. In tal caso i costruttori prevedono di poter omettere il pattern.

```
Ris ris = new Ris(controlTerm, dominio, filtro);
```

1.2.1 Constraints sui Ris

I `Ris` di JSetL permettono, come tutti gli `LSet`, di esprimere vincoli sotto forma di oggetti di tipo `Constraint`. Questi vincoli sono principalmente gli stessi degli `LSet` generici di JSetL. Vediamo ora ciascuno dei vincoli principali con un breve esempio di uso.

- `Constraint` di uguaglianza, creato con il metodo `eq`, (e analogamente il vincolo di diverso, creato col metodo `neq`) permette di esprimere l'uguaglianza insiemistica di un `Ris` con un altro `LSet`(possibilmente anche un altro `Ris`). Il seguente esempio crea un `Ris` con un dominio non specificato, un filtro che accetta solo i control term dispari e un pattern che raddoppia il control term. Infine pone il `Ris` uguale all'insieme $res = \{6, 2, 26\}$ e risolve il vincolo. Di conseguenza viene dedotto che nel dominio debbano essere presenti i numeri 3, 1 e 13.

  ```
  IntLVar x = new IntLVar();
  IntLSet dom = new IntLSet();
  Ris ris = new Ris(x, dom, x.mod(2).eq(1), x.mul(2));
  IntLSet res = IntLSet.empty().ins(26).ins(2).ins(6);
  ```

```
SolverClass solver = new SolverClass();
solver.add(ris.eq(res));
System.out.println(solver.check()); // true
dom.output(); // ? = {3,1,13|_C} i.e. {3, 1, 13} ∪ _C
```

- `Constraint` di appartenenza insiemistica, creato con il metodo `contains`(e analogamente il vincolo di non appartenenza, creato col metodo `ncontains`) permette di esprimere l'appartenenza in un oggetto(sia logico che non) al `Ris`. Il seguente esempio usa un `Ris` come quello del punto precedente ma con un dominio definito, l'insieme $dom = \{2, 5, 9\}$, chiede poi al solver se 10 appartiene all'insieme e ottiene risposta positiva.

```
IntLVar x = new IntLVar();
IntLSet dom = IntLSet.empty().ins(2).ins(5).ins(9);
Ris ris = new Ris(x, dom, x.mod(2).eq(1), x.mul(2));
SolverClass solver = new SolverClass();
solver.add(ris.contains(10));
solver.check(); // true
```

Capitolo 2

Usi dei RIS

2.1 Usare i RIS come funzioni

L'implementazione dei RIS in JSetL permette di utilizzare i benefici della programmazione dichiarativa all'interno di programmi imperativi Java. I RIS possono denotare insiemi di coppie di variabili prese da insiemi diversi, ovvero relazioni su insiemi. Le funzioni parziali sono spesso modellate come particolari relazioni in cui ad ogni elemento del dominio corrisponde al più un elemento del codominio. I Ris possono essere passati come parametri a funzioni Java. Possiamo dunque utilizzare i RIS come struttura dati per effettuare il passaggio di una funzione(matematica) come parametro di una funzione Java.

2.1.1 Passaggio di funzioni in Java

In questa sezione è presente un esempio di programma in Java per passare funzioni che vengono poi applicate agli elementi di un array di `Integer` e restituendo l'array così costruito. Per poterlo fare è necessario creare un'interfaccia funzionale, in questo caso chiamata `FunctionInt` con un unico metodo `apply` che prende un `int` e restituisce un `int`. Questa interfaccia viene poi implementata da una inner class anonima con il codice della funzione e l'oggetto così ottenuto viene passato al metodo `apply` che prende un

`FunctionInt` e un array di `Integer` e restituisce l'array delle immagini. File: FunctionInt.java

```
package JSetL.prove.RIS;

public interface FunctionInt {

   public int apply(int arg);

}
```

File: InnerFunctionApply.java

```
package JSetL.prove.RIS;

import java.util.Arrays;

public class InnerFunctionApply {

   public static Integer[] apply(FunctionInt function, Integer[] arr) {

      Integer[] images = new Integer[arr.length];
      for(int i = 0; i < arr.length; ++i) {
         images[i] = function.apply(arr[i]);
      }
      return images;

   }
   public static void main(String[] args) {
      Integer[] arguments = {1,2,3,4,7,-1,0};
      Integer[] results = apply(new FunctionInt() {
         @Override
         public int apply(int x) {
            return x + x * x;
         }
      }, arguments);

      System.out.println(Arrays.asList(results));
   }
```

```
}
```

2.1.2 Passaggio di funzioni coi RIS

La stessa cosa vista prima si può fare utilizzando i `Ris`, senza bisogno di definire interfacce e di usare inner class anonime. Per definire la funzione si crea un oggetto di tipo `Ris` con filtro sempre verificato e dominio variabile e in cui la funzione vera e propria è specificata dal pattern: il ris è l'insieme delle coppie $(x, x + x^2)$, e denota dunque la funzione $f(x) = x + x^2$. Il ris così costruito è passato insieme all'array di `Integer` alla funzione `apply` che per ogni elemento dell'array applica la funzione denotata dal `Ris`. Per applicare la funzione si sfruttano le caratteristiche del solver di JSetL: si impone che la coppia `(arr[i],y)`, con `y` non inizializzata, appartenga al `Ris`. L'unificazione calcolata dal solver associa il corretto valore a `y`, che viene poi estratto con il metodo `getValue()`.

```
package JSetL.prove.RIS;

import java.util.Arrays;
import JSetL.*;

public class RisApply {

   public static Integer[] apply(Ris function, Integer[] arr) {
      Integer[] images = new Integer[arr.length];
      for(int i = 0; i < arr.length; ++i) {
         IntLVar y = new IntLVar();
         function.contains(LList.mkPair(arr[i], y)).check();
         images[i] = y.getValue();
      }
      return images;
   }
   public static void main(String[] args) {
      Integer[] arguments = {1,2,3,4,7,-1,0};
      IntLVar x = new IntLVar();
```

```
      Integer[] results = apply(new Ris(x, new IntLSet(), Constraint.TRUE,
           LList.mkPair(x, x.sum(x.mul(x)))),arguments);

      System.out.println(Arrays.asList(results));
   }

}
```

Di fatto l'uso del `Ris` per passare la funzione $f(x) = x + x^2$ rende il codice più corto e, per chi conosce bene JSetL, più semplice.

2.1.3 Passaggio di funzioni con Lambda Expressions

Bisogna comunque notare che da Java 8 sono state introdotte le lambda expression come zucchero sintattico per semplificare la scrittura delle inner class anonime usate per passare funzioni. Il codice non risulta comunque più corto rispetto a quello che usa JSetL in quanto è comunque necessario creare una interfaccia funzionale.

File: FunctionInt.java

```
package JSetL.prove.RIS;

public interface FunctionInt {

   public int apply(int arg);

}
```

File: LambdasApply.java

```
package JSetL.prove.RIS;

import java.util.Arrays;

public class LambdasApply {

   public static Integer[] apply(FunctionInt function, Integer[] arr) {
```

```
      Integer[] images = new Integer[arr.length];
      for(int i = 0; i < arr.length; ++i) {
         images[i] = function.apply(arr[i]);
      }
      return images;

   }
   public static void main(String[] args) {
      Integer[] arguments = {1,2,3,4,7,-1,0};
      Integer[] results = apply((x) -> x + x*x , arguments);

      System.out.println(Arrays.asList(results));
   }

}
```

2.1.4 Filtrare un array in Java

I Ris contengono al loro interno un filtro. Esso può essere sfruttato in maniera semplice per filtrare gli elementi di un array: è sufficiente creare un `Constraint` di tipo `contains` e usare il metodo `check` per risolverlo: se il risultato è `true` allora l'elemento rispetta il filtro del `Ris`, altrimenti no. File: RisFilter.java

```
package JSetL.prove.RIS;

import java.util.Arrays;
import java.util.LinkedList;
import JSetL.*;

public class RisFilter {
   public static Object[] filter(Ris function, Integer[] arr) {
      LinkedList<Integer> filtered = new LinkedList<>();
      for(Integer i : arr)
         if(function.contains(new IntLVar(i)).check())
            filtered.add(i);
```

```
      return filtered.toArray();
   }
   public static void main(String[] args) {
      Integer[] arguments = {1,2,3,4,7,-1,0};
      IntLVar x = new IntLVar();

      Object[] results = filter(new Ris(x, new IntLSet(), x.mod(2).eq(0)),
          arguments);
      System.out.println(Arrays.asList(results));
   }
}
```

2.2 Usare i Ris come relazioni

Visto che sia il control term che, soprattutto, il pattern di un Ris possono essere una coppia, e in generale una tupla, è semplice esprimere relazioni insiemistiche coi Ris.

2.2.1 Un esempio: il dizionario Italiano-Inglese

Mostriamo ora un esempio di programma in riga di comando che permette di inserire traduzioni arbitrarie di parole italiane ed inglesi e poi effettuare traduzioni parola per parola da italiano a inglese e viceversa. Attraverso la deduzione automatica del dominio, e questo è il lato più interessante, vengono compilati contemporaneamente sia l'insieme delle parole note italiane, che quello delle parole note inglesi che quello delle traduzioni note. Il meccanismo è realizzato attraverso un Ris, che è anche l'unico oggetto che viene passato come parametro come dizionario di traduzione.

File: ThesaurusExample.java

```
package JSetL.prove.RIS;

import java.io.BufferedReader;
```

```
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

import JSetL.Constraint;
import JSetL.Failure;
import JSetL.LList;
import JSetL.LSet;
import JSetL.LVar;
import JSetL.Ris;
import JSetL.SolverClass;

public class ThesaurusExample {
   private static void translate(Ris thesaurus, LVar italian, LVar english
       ) throws Failure{
      thesaurus.contains(LList.mkPair(italian, english)).solve();
   }

   private static String translateFromItalian(Ris thesaurus, String
       italian) throws Failure{
      LVar italianvar = new LVar("italian", italian);
      LVar englishvar = new LVar("english");
      translate(thesaurus, italianvar, englishvar);
      return (String) englishvar.getValue();
   }

   private static String translateFromEnglish(Ris thesaurus, String
       english) throws Failure{
      LVar italianvar = new LVar("italian");
      LVar englishvar = new LVar("english", english);
      translate(thesaurus, italianvar, englishvar);
      return (String) italianvar.getValue();
   }

   private static void translate(Ris thesaurus, String italian, String
       english) throws Failure{
      LVar italianvar = new LVar("italian", italian);
      LVar englishvar = new LVar("english", english);
```

```
    translate(thesaurus, italianvar, englishvar);

}

private static void printWelcome() {
    System.out.println("Welcome to babeldog -cit.");
}

private static int EXIT_CHOICE = 9;
private static void printMenu() {
    System.out.println("###MENU###");
    System.out.println("1. Insert a translation");
    System.out.println("2. Translate a sentence from italian(word by
        word)");
    System.out.println("3. Translate a sentence from english(word by
        word)");

    System.out.println("4. Print all translations known");
    System.out.println("5. Print all italian words known");
    System.out.println("6. Print all english words known");
    System.out.println("9. Exit");

}

private static int getChoice() {
    System.out.println("Enter your choice, between 1 and 6");
    Scanner scanner = new Scanner(System.in);
    do {
        int choice = Integer.parseInt(scanner.nextLine());
        if(choice >= 1 && choice <= 6 || choice == EXIT_CHOICE) {
            return choice;
        }
        else
            System.out.println("You did not enter a valid choice. Retry
                please.");
    } while(true);
```

```
}

private static void insertTranslation(Ris translations) throws Failure
    , IOException{
   System.out.println("You will now enter a correct (italian, english)
       translation pair.");
   BufferedReader br = new BufferedReader(new InputStreamReader(System.
       in));
   System.out.print("Enter a single italian word please: ");
   String italianWord = br.readLine();

   System.out.print("Enter a single english word please: ");

   String englishWord = br.readLine();

   translate(translations, italianWord, englishWord);
}

private static void translateSentenceFromItalian(Ris translations)
    throws Failure {
   System.out.println("You will now enter an italian sentence in a
       single line");
   System.out.print("Italian: ");
   Scanner scanner = new Scanner(System.in);

   String italianSentence = scanner.nextLine();

   String[] italianWords = italianSentence.split(" ");
   StringBuilder builder = new StringBuilder();
   for(String word : italianWords)
      builder.append(translateFromItalian(translations, word) + " ");

   String englishSentence = builder.toString();

   System.out.println("English: " + englishSentence);
}

private static void translateSentenceFromEnglish(Ris translations)
```

```
    throws Failure {
   System.out.println("You will now enter an english sentence in a
       single line");
   System.out.print("English: ");
   Scanner scanner = new Scanner(System.in);

   String englishSentence = scanner.nextLine();

   String[] englishWords = englishSentence.split(" ");
   StringBuilder builder = new StringBuilder();
   for(String word : englishWords)
      builder.append(translateFromEnglish(translations, word) + " ");

   String italianSentence = builder.toString();

   System.out.println("Italian: " + italianSentence);
}

private static void showTranslations(Ris translationThesaurus) throws
    Failure {
   System.out.println(translationThesaurus.expand());
}

private static void showWords(LSet words) {
   System.out.println(words);
}


private static void executeChoice(int choice, Ris translationThesaurus,
     LSet italianWords, LSet englishWords) throws Failure, IOException
    {
   switch(choice) {
   case 1: insertTranslation(translationThesaurus); break;
   case 2: translateSentenceFromItalian(translationThesaurus); break;
   case 3: translateSentenceFromEnglish(translationThesaurus); break;
   case 4: showTranslations(translationThesaurus); break;
   case 5: showWords(italianWords); break;
   case 6: showWords(englishWords); break;
```

```
      default:
         throw new IllegalArgumentException();
      }
   }
   public static void main(String[] args) throws Failure, IOException {
      /* initializing dictionaries */
      LSet translations = new LSet("synonyms");
      LSet italianWords = new LSet("italian_words");
      LSet englishWords = new LSet("english_words");
      LVar italian = new LVar("italian");
      LVar english = new LVar("english");
      Constraint constraint = italian.in(italianWords).and(english.in(
          englishWords));
      Ris translations_thesaurus = new Ris(LList.mkPair(italian, english),
           translations, constraint);

      // welcoming
      printWelcome();


      do {
         printMenu();
         int choice = getChoice();
         if(choice == ThesaurusExample.EXIT_CHOICE)
            break;
         executeChoice(choice, translations_thesaurus, italianWords,
             englishWords );
      }while(true);

      System.out.println("Execution ended correctly.");
   }

}
```

Mostriamo ora un esempio di dialogo del programma per chiarirne meglio il funzionamento.

```
 Welcome to babeldog -cit.
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
1
You will now enter a correct (italian, english) translation pair.
Enter a single italian word please: ciao
Enter a single english word please: hello
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
1
You will now enter a correct (italian, english) translation pair.
Enter a single italian word please: mondo
Enter a single english word please: world
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
2
You will now enter an italian sentence in a single line
```

```
Italian: ciao mondo
English: hello world
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
3
You will now enter an english sentence in a single line
English: hello world
Italian: ciao mondo
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
4
{[mondo,world],[ciao,hello]}
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
5
{ciao,mondo|_?}
###MENU###
```

```
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
6
{hello,world|_?}
###MENU###
1. Insert a translation
2. Translate a sentence from italian(word by word)
3. Translate a sentence from english(word by word)
4. Print all translations known
5. Print all italian words known
6. Print all english words known
9. Exit
Enter your choice, between 1 and 6
9
Execution ended correctly.
```

Capitolo 3

Nuovi vincoli per i Ris

Oltre ad utilizzare i quattro vincoli base dei `Ris`, `eq, neq, contains` e `ncontains` mi sono occupato di implementare due nuovi vincoli insiemistici, già presenti per gli `LSet`, secondo le regole di riscrittura di (5). I nuovi vincoli permettono di esprimere la disgiunzione fra insiemi e l'unione. Entrambi questi nuovi vincoli sono stati testati, e il mio lavoro si è concentrato su quelli. Nella terza sezione vedremo però che, grazie ai meccanismi già in atto in JSetL e a questi due nuovi vincoli, altri vincoli interessanti sugli insiemi dovrebbero essere automaticamente supportati, grazie alla robustezza delle regole di riscrittura usate per tali vincoli nel caso di `LSet`.

3.1 Vincolo di disgiunzione

Due insiemi si dicono disgiunti se non hanno alcun elemento in comune.

$$A||B <=> \forall x \in A : x \notin B$$

Per esempio, sia $D = \{1, 2, 3, 4, 5, 6\}$, $B = \{x \in D | x \bmod 2 = 0\}$ e $A = \{x \in D | x \bmod 2 = 1\}$, allora abbiamo $A||B$.
Per implementare il vincolo di disgiunto ho implementato le seguenti tre regole di riscrittura del vincolo.

- $$\{D|F \bullet P\}||\{\} -> true$$

Questo perché ogni insieme è disgiunto con l'insieme vuoto.

- $$\{D|F \bullet P\}||\{x|A\} -> x \notin \{D|F \bullet P\} \wedge \{D|F \bullet P\}||A$$

  Affinché un insieme contenente l'elemento x e avente resto A sia disgiunto con un Ris bisogna che x non appartenga al Ris e, ricorsivamente, l'insieme A sia disgiunto col Ris.

- $$\{D|F \bullet P\}||\{E|G \bullet Q\} -> irriducibile$$

  In quanto i due Ris sono variabili e in particolare hanno dominio variabile e l'unificazione $D = E = \{\}$ verifica il vincolo. Queste regole di riscrittura sono state applicate sia al caso in cui uno solo degli operandi fosse un `Ris` (e l'altro un `LSet`), sia al caso in cui fossero entrambi `Ris`. Per via della commutatività del vincolo nel caso in cui l'operando di sinistra non fosse un `Ris` ho scambiato gli operandi prima di procedere con la riscrittura.

3.1.1 Uso del disgiunto nei Ris di JSetL

Il metodo per creare un `Constraint` per il vincolo di disgiunzione è lo stesso per i `Ris` e per gli `LSet`: `disj`. Il seguente esempio è uno dei test usati per verificare la correttezza dell'implementazione quando l'operando di destra è l'insieme vuoto.

```
// {x:D | x = x @ x} || void
LSet lvoid = LSet.empty();
IntLVar x = new IntLVar("x");
LSet d = new LSet("d");
Ris ris = new Ris(x, d, x.eq(x));

SolverClass solver = new SolverClass();
solver.add(ris.disj(lvoid));
```

```
assertEquals(true, solver.check());
```

Il seguente esempio invece è un test per la seconda regola e verifica che i due ris che denotano gli intervalli di interi [1..10] e [8..20] non sono disgiunti.

```
// {x:D | x = x @ x} || {y:B | y= y @ y}

IntLVar x = new IntLVar("x");
IntLSet d = new IntLSet(1,10);
Ris ris = new Ris(x, d, x.eq(x));
assertEquals(true, ris.isGround());

IntLVar y = new IntLVar("y");
IntLSet b = new IntLSet(8,20);

Ris ris2 = new Ris(y, b, y.eq(y), y);

SolverClass solver = new SolverClass();
solver.add(ris.disj(ris2));
assertEquals(false, solver.check());
```

Questo ultimo esempio testa l'implementazione della terza regola di riscrittura, quella che tiene il vincolo come irriducibile in quanto la soluzione `d = b = {}` è possibile e corretta.

```
// {x:D | x = x @ x} || {y:B | y= y @ y}

IntLVar x = new IntLVar("x");
IntLSet d = new IntLSet();
Ris ris = new Ris(x, d, x.eq(x));

IntLVar y = new IntLVar("y");
IntLSet b = new IntLSet();

Ris ris2 = new Ris(y, b, y.eq(y), y);

SolverClass solver = new SolverClass();
```

```
solver.add(ris.disj(ris2));
assertEquals(true, solver.check());
```

3.2 Vincolo di unione

Il vincolo di unione è un vincolo ternario. Denotiamo con $union(A, B, C)$ un vincolo che è soddisfatto se e soltanto se $C = A \cup B$, ovvero

$$(\forall x \in C : x \in A \vee x \in B) \wedge (\forall y \in A : y \in C) \wedge (\forall z \in B : z \in C).$$

Poiché il vincolo è ternario ed ha un numero di casi elevato è opportuno considerare un ulteriore vincolo fittizio, $\mathrm{T}(A) = B$ definito come segue:

$$\mathrm{T}(x) = \begin{cases} x, & \text{se } x \text{ non è un RIS} \\ \{P(x)|\{D|F \bullet P\}, & \text{se } x_i \text{ è un RIS e } F(x_i) \text{ vale} \\ \{D|F \bullet P\}, & \text{se } x_i \text{ è un RIS e } F(x_i) \text{ non vale} \end{cases}$$

Questa funzione di fatto "espande" il Ris un elemento alla volta. A questo punto usiamo la funzione Tau per implementare il vincolo di unione, con due sole regole di riscrittura.

- $union(A, B, C) \rightarrow$ *irriducibile* se A, B e C sono tutti `Ris` variabili, ovvero con dominio variabile.
- $union(A, B, C) \rightarrow union(\mathrm{T}(A), \mathrm{T}(B), \mathrm{T}(C))$ altrimenti. Dopo aver applicato questa regola di fatto tutti e tre gli argomenti del vincolo saranno dei normali `LSet` pertanto si applicherà la regola già presente in JSetL, con la possibilità di ricadere in questa regola(o nella precedente) ricorsivamente.

3.2.1 Uso dell'unione nei Ris di JSetL

Il metodo per creare `Constraint` per il vincolo di unione coi `Ris` è lo stesso degli `LSet`: `union`. Il metodo ha la seguente sintassi: `ris.union(set1,set2)`

e corrisponde al vincolo $union(set1, set2, ris)$, ovvero $ris = set1 \cup set2$. Quello che segue è uno dei test per il vincolo di unione. Si crea un `Ris` che denota l'insieme $\{x \in \{1,2,3\} | x \bmod 2 = 0\}$ e si verifica che esso sia l'unione dell'insieme vuoto con il singoletto 2.

```
IntLSet domain = IntLSet.empty().ins(1).ins(2).ins(3);
IntLVar x = new IntLVar();
Ris ris = new Ris(x, domain, x.mod(2).eq(0));
assertEquals(true, ris.union(LSet.empty(), IntLSet.empty().ins(2)).check()
    );
```

In questo altro test si prova il funzionamento dell'unione con tre `Ris` ognuno con il proprio dominio completamente specificato. I primi due `Ris` prendono solo gli elementi pari dai rispettivi domini mentre il terzo prende gli elementi pari e diversi da 900. Si verifica che `ris3` è il risultato dell'unione di `ris1` con `ris2`, in quanto 902 non fa parte dell'insieme a causa del filtro.

```
IntLSet domain1 = IntLSet.empty().ins(1).ins(0).ins(2).ins(3).ins(1000);
IntLVar x1 = new IntLVar();
Ris ris1 = new Ris(x1, domain1, x1.mod(2).eq(0));

IntLSet domain2 = IntLSet.empty().ins(1001).ins(0).ins(2).ins(101);
IntLVar x2 = new IntLVar();
Ris ris2 = new Ris(x2, domain2, x2.mod(2).eq(0));

IntLSet domain3 = IntLSet.empty().ins(1000).ins(0).ins(2).ins(101).ins
    (900);
IntLVar x3 = new IntLVar();
Ris ris3 = new Ris(x3, domain3, x3.mod(2).eq(0).and(x3.neq(900)));

assertEquals(true, ris1.union(ris2,ris3).check());
```

Infine quest'ultimo esempio mostra il funzionamento del vincolo nel caso di `Ris` con domini non completamente specificati e `LSet`. Dall'ultima riga vediamo che il procedimento di risoluzione del vincolo ha effettivamente dedotto che `domain3` debba contenere anche 4 affinché il vincolo sia soddisfatto.

```
IntLSet domain = new IntLSet().ins(4);
```

```
IntLVar x = new IntLVar();
Ris ris = new Ris(x, domain, x.mod(2).eq(0));

IntLSet domain3 = new IntLSet().ins(1000).ins(0).ins(2).ins(101).ins(902);
IntLVar x3 = new IntLVar();
Ris ris3 = new Ris(x3, domain3, x3.mod(2).eq(0).and(x3.neq(900)));
assertEquals(true, ris.union(ris3, new IntLSet().ins(6)).check());
assertEquals("{4,902,2,0,6,1000|_C}", domain.toString());
```

3.3 Vincoli derivati

Ci sono almeno due vincoli comuni sugli insiemi che non ho trattato direttamente: il vincolo di sottoinsieme e il vincolo di intersezione. In realtà JSetL riscrive immediatamente questi vincoli in termini dei due vincoli precedenti secondo le seguenti regole.

- $S_1 \subset S_2 \rightarrow S_1 \cup S_2 = S_2$
- $S_3 = S_1 \cap S_2 \rightarrow S_1 = S_3 \cup S_{aux1} \wedge S_2 = S_3 \cup S_{aux2} \wedge S_{aux1} \parallel S_{aux2}$

Dunque i vincoli di sottoinsieme e di intersezioni vengono espressi in termini di uguaglianza insiemistica, unione e disgiunzione e pertanto dovrebbero funzionare correttamente.

Capitolo 4

Riscrittura dei vincoli di inequalità

Quando un `LSet` non inizializzato si trovano in un vincolo di inequalità esso non viene riscritto e viene considerato irriducibile e risolto. Questo comportamento è quasi sempre corretto se non fosse che porta a inconsistenze con altri vincoli in forma irriducibile che vengono considerati risolti in quanto l'unificazione con l'insieme vuoto soddisfa il vincolo. Se però nel constraint store è presenta anche un vincolo che impone che quello stesso insieme non possa essere l'insieme vuoto allora la risposta data dal solver potrebbe essere corrette. Il problema si presenta sia senza i `Ris` che con, in maniera diversa. Per chiarezza esponiamo il problema e cosa bisogna fare per risolverlo in maniera separata per gli `LSet` e per i `Ris`, anche se la soluzione effettivamente adottata nel codice tiene conto di entrambi i problemi contemporaneamente.

4.1 Problemi coi vincoli su LSet

Mostriamo innanzitutto un esempio in cui il problema si manifesta, facendo fallire il test per poi spiegarne le cause e la soluzione.

```
LSet A1 = new LSet();
LSet A2 = new LSet();
```

```
LSet B = new LSet();
LSet C = new LSet();
SolverClass solver = new SolverClass();
solver.add(A1.union(B, C));
solver.add(A2.union(B, C));
solver.add(A1.neq(A2));
assertEquals(false, solver.check());
```

Il test fallisce perché i vincoli vengono analizzati dal solver uno alla volta e tutti sono lasciati in forma risolta. In particolare il problema viene generato dal fatto che il vincolo `A1.neq(A2)` non venga riscritto e dunque l'informazione sulla diversità di `A1` e `A2` non venga usata per dedurre che non possono entrambi essere il risultato dell'unione degli insiemi `B` e `C`. Il problema si verifica solo con l'unione con gli `LSet` e per risolverlo è sufficiente applicare, ai vincoli di inequalità che coinvolgono `LSet` coinvolti in vincoli di unione, la seguente riscrittura. In particolare, se l'argomento di destra della inequalità è un LSet si applica la seguente regola (altrimenti si inverte l'ordine degli argomenti e poi si applica comunque questa regola).

$$A \neq t \rightarrow x \in A \wedge x \notin B \vee x \in B \wedge x \notin A \vee A = \{\} \wedge x \neq \{\}.$$

Le tre riscritture in or sono da effettuarsi in maniera non deterministica. La terza opzione può sembrare ridondante ma è necessaria nel caso in cui t non sia un `LSet`: in tal caso la seconda opzione fallirebbe e si perderebbe la soluzione con $A = \{\}$.

4.2 Problemi coi vincoli sui Ris

Anche in questo caso mostraimo innanzitutto un esempio in cui il problema si manifesta per poi spiegarne causa e soluzione.

```
LVar x = new LVar();
LSet dom = new LSet();
Ris ris = new Ris(x, dom, x.eq(x));
```

```
SolverClass solver = new SolverClass();

solver.add(ris.eq(LSet.empty()));
solver.add(dom.neq(LSet.empty()));
assertEquals(false, solver.check());
```

Per come è definito è immediato che valga `ris` = `dom`. Il primo vincolo aggiunto al `solver` richiede che `ris` sia l'insieme vuoto mentre il secondo richiede che il dominio non lo sia. Questi due vincoli sono inconsistenti ma tale incoerenza non viene trovata dal solver e pertanto il test fallisce. Il motivo è che il secondo vincolo viene lasciato in forma non risolta e pertanto l'incongruenza non viene scoperta. In questo caso bisogna applicare la regola vista nella sezione precedente per riscrivere i vincoli di `neq` che coinvolgono i domini dei `Ris`. Attualmente non c'è il supporto per `Ris` che hanno per dominio un altro `Ris` ma per rendere la regola robusta bisogna che gli insiemi considerati siano i domini più "interni" dei `Ris`.

4.3 Soluzione adottata

La soluzione adottata per risolvere questi due problemi è la seguente: dopo ogni riscrittura completa dei vincoli nel constraint store si calcolano gli insiemi di tutti gli insiemi coinvolti nei vincoli di unione e di tutti i domini più interni dei `Ris`. Dopodichè per ogni disuguaglianza si applica la regola di riscrittura descritta nella prima sezione se uno dei due argomenti appartiene a uno dei due insiemi appena calcolati. Dopodiché si fa in modo che il solver faccia un altro giro completo dei vincoli per risolvere quelli appena generati e, possibilmente, trovare le inconsistenze. Con questa soluzione gli esempi delle sezioni precedenti funzionano tutti correttamente.

Capitolo 5

Un'estensione ai RIS: variabili esistenziali nel filtro e nel pattern

5.1 Le istanze del control term

Nella prima implementazione dei Ris, presentata in (6), il pattern, quando diverso dal control term, è una `IntLVar` e infatti i metodi usati per definire l'espressione del pattern, ad esempio `mul` e `sum`, restituiscono una `IntLVar` che contiene al proprio interno i vincoli associati che la legano al control term. Il control term però deve essere una variabile esistenziale, nel senso che bisogna crearne una nuova istanza `d` ogni volta che si deve verificare un vincolo e questa istanza `d` viene usata dai metodi privati di `Ris` `P(d)` e `F(d)` per generare le istanze di filtro e pattern legate a `d`. Questo meccanismo assicura che la nuova istanza del control term venga usata sia dal filtro che dal pattern. Quessto approccio permette di far funzionare correttamente i Ris descritti in (4).

5.2 Variabili esistenziali aggiuntive

Una possibile estensione ai Ris consiste nel prevedere che possano essere presenti variabili esistenziali nel pattern e nel filtro del Ris, dove con esistenziali intendiamo soggette allo stesso trattamento del control term. Vogliamo dunque essere in grado di risolvere il seguente vincolo.

$$D = \{2\} \wedge R = \{[2, a], [1, b], [2, c]\} \wedge \{[2, a], [2, c]\} = \{X : D | \exists Y.[X, Y] \in R \bullet [X, Y]\}$$

Questo vincolo prevede una variabile esistenziale nel filtro, che compare anche nel pattern. Per distinguere le variabili esistenziali JSetL prevede che esse abbiano un nome che inizia con la stringa `_I`. Tuttavia il problema che impediva a questo vincolo sui Ris non previsti di funzionare era connesso al fatto che, pur creando la variabile `Y` con un nome che la rendesse esistenziale comunque non si arrivava al risultato corretto in quanto ogni volta che essa compare ne viene generata una nuova istanza, e le istanze non sono collegate fra loro. Dunque la nuova istanza di `Y` nel pattern non è legata in alcun modo a quella nel filtro e ciò non è chiaramente quello che si intende. Per risolvere il problema sfruttando al contempo il meccanismo già realizzato in (6) si è aggiunta una proprietà privata alla classe `Ris` contenente una mappa fra le variabili che compaiono nel filtro del Ris e la loro ultima istanza. Ogni volta che si trova una variabile esistenziale nella valutazione del filtro o del pattern si crea una nuova istanza se la mappa non la contiene già oppure la si ottiene dalla mappa. Questo meccanismo assicura che filtro e pattern condividano le stesse istanze di variabili esistenziali. Poiché ogni volta che si genera il pattern si genera prima il filtro, la mappa viene ripulita con il metodo `clear` appena prima di generare il filtro, questo assicura che vengano sempre create le nuove istanze che sono necessarie. Attualmente vengono aggiunte alla mappa, e dunque trattate correttamente, solo le variabili logiche che appaiono al primo livello di profondità del filtro, ovvero solo le variabili che compaiono come operando di un `AConstraint` contenuto nel filtro. Pertanto al momento è necessario che ogni variabile esistenziale compaia nel filtro in un vincolo che la veda come primo, secondo o terzo operando. Per

risolvere questo problema sarà necessario controllare il filtro in profondità e controllare il pattern.
Affinché tutto funzioni è necessario prevedere un filtro che permetta al solver di assegnare un valore alla variabile esistenziale. Bisogna cioè specificare, direttamente o indirettamente, un dominio per essa all'interno del filtro. Il modo più semplice per farlo è aggiungere al filtro un vincolo del tipo `y.in(R)` con `y` variabile esistenziale e `R` insieme completamente specificato.

5.3 Un programma di esempio

Vediamo ora come si può risolvere il vincolo visto nella sezione precedente sfruttando il supporto alle variabili esistenziali. Inizialmente si creano le tre coppie $[2, a], [1, b], [2, c]$, poi si crea l'insieme `R` che le contiene tutte e tre e un dominio `D` per il control term `x` che contiene solo il numero 2. Si crea dunque l'insieme estensionale $\{[2, a], [2, c]\}$ e lo si assegna alla variabile `sol`. Viene poi create una variabile esistenziale `y`, che ha dunque nome `_I`. Si crea una istanza del solver e si crea la coppia $[x, y]$ attraverso il metodo statico `LList.makePair(x,y)`. Si crea dunque il `Ris` corrispondente a quello dell'esempio e si aggiunge il vincolo di uguaglianza `sol.eq(ris)`. Infine si verifica che il solver riesca a risolvere il vincolo. Bisogna notare che in questo esempio non specifichiamo un dominio per la variabile esistenziale nel modo visto nella sezione precedente, ma specifichiamo un dominio per essa in maniera indiretta: specifichiamo infatti il dominio per la coppia $[x, y]$ e dunque in automatico la `y` potrà assumere solo i valori dei secondi elementi delle coppie contenute in `R`, che è un insieme completamente specificato.

```
LList apair = LList.mkPair(2, "a");
LList bpair = LList.mkPair(1, "b");
LList cpair = LList.mkPair(2, "c");

LSet R = LSet.empty().ins(apair).ins(bpair).ins(cpair);
IntLSet D = IntLSet.empty().ins(2);
```

```
LSet sol = LSet.empty().ins(cpair).ins(apair);
IntLVar x = new IntLVar();
LVar y = new LVar("_I");
SolverClass solver = new SolverClass();
LList pairxy = LList.mkPair(x, y);
Ris ris = new Ris(x, D, R.contains(pairxy).and(y.eq(y)),pairxy);

solver.add(sol.eq(ris));
assertEquals(true, solver.check());
```

Capitolo 6

Conclusioni e sviluppi futuri

In questa tesi è stato descritto come si possano usare i `Ris` per definire funzioni parziali e relazioni fra insiemi e passare tali oggetti come parametri a metodi Java. Successivamente è stato svolto un lavoro di arricchimento della libreria JSetL col supporto a nuove operazioni insiemistiche sui `Ris`: l'unione e la disgiunzione. Si è poi risolto il problema dei vincoli $\neq$ irrisolti che portavano a inconsistenze con altri vincoli irrisolti e tale problema è stato affrontato anche per i `Ris`. Ci si è poi occupati di iniziare ad estendere il supporto ai `Ris` con variabili esistenziali, che esula da quanto definito originariamente in (4). Sarebbe opportuno verificare che i vincoli di sottoinsieme e di intersezione per i `Ris` siano gestiti correttamente da JSetL, dato che internamente vengono riscritti usando solo vincoli ora implementati, e per farlo scrivere una serie di test che fungerebbero anche da ulteriori test per i vincoli implementati in questa tesi. Sarebbe inoltre opportuno migliorare il supporto alle variabili esistenziali nel filtro e nel pattern dei `Ris` togliendo il vincolo che esse debbano apparire come argomenti di un `AConstraint` nel filtro per essere riconosciute come variabili esistenziali. Sarebbe inoltre opportuno prevedere un modo più elegante di quello utilizzato attualmente per definire variabili esistenziali, magari attraverso un metodo statico delle classi `LVar` e `IntLVar`. Un'altra opportunità di miglioramento consiste nell'aggiungere e testare il supporto alla classe `Pair` come control term o pattern dei

`Ris`, che al momento è solo parziale. Sarebbe inoltre opportuno permettere di creare `Ris` con un control term o pattern di tipo `LSet` o `IntLSet` per poter ad esempio calcolare l'insieme potenza di un insieme completamente specificato R con un insieme intensionale del tipo

$$\{S : D \,|\, S \subset R \bullet S\}$$

con D dominio completamente variabile. Infine, uno degli approfondimenti più interessanti consiste nel verificare il supporto ai `Ris` ricorsivi, che attualmente non funzionano correttamente in JSetL. Migliorando l'implementazione dovrebbe essere possibile calcolare tramite i `Ris` funzioni ricorsive come il fattoriale, utilizzando un `Ris` dalla forma seguente.

$$F = \{X : D \,|\, \exists X_1, F, Y.([X_1, F] \in F \wedge X > 0 \wedge Y = XF \wedge X_1 = X - 1) \vee (X = 0 \wedge Y = 1) \bullet [X, Y]\}$$

Bibliografia

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

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

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

[4] Maximiliano Cristià e Gianfranco Rossi
A Decision Procedure for Restricted Intensional Sets

[5] Maximiliano Cristià e Gianfranco Rossi
Note interne su vincoli di unione e disgiunzione

[6] Andrea Guerra
Estensione della libreria Java JSetL con gli Insiemi Intensionali Ristretti

[7] ISO/IEC 13568
Information technology — Z formal specification notation — Syntax, type system and semantics

[8] Kim Marriott and Peter J. Stuckey with contributions from Leslie De Koninck and Horst Samulowitz
A MiniZinc Tutorial

[9] function* - Javascript — MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*