UNIVERSITÀ
DI PARMA

DIPARTIMENTO DI
SCIENZE MATEMATICHE, FISICHE E
INFORMATICHE

Corso di Laurea in Informatica

Tesi di Laurea

Implementazione e valutazione di regole di riscrittura per vincoli insiemistici nella libreria Java JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Federico Marchi

Anno Accademico 2019/2020

A Stefania, Andrea, Francesco, Matteo, i miei nonni, Maria Vittoria e a tutti quelli che mi vogliono bene ma non riesco a citare.

Indice

Introduzione

La programmazione dichiarativa (DP) è un paradigma di programmazione che prevede di descrivere la soluzione del problema, anziché il procedimento che porta ad ottenerla.

Un particolare tipo di DP è la *programmazione a vincoli*, di cui un'interessante applicazione è la risoluzione di vincoli insiemistici.

Dato un linguaggio di programmazione a vincoli su insiemi, la scelta su quali operazioni insiemistiche dovrebbero essere *primitive* (o built-in) dipende da vari criteri quali potere espressivo, completezza, efficacia ed efficienza. Minimizzare il numero di operazioni primitive ha il vantaggio di ridurre il numero di tipi di vincoli da gestire e, sperabilmente, semplificare il linguaggio e la sua implementazione. In contrapposizione, aumentare il numero di operazioni primitive favorisce l'efficienza.

JSetL è una libreria Java, sviluppata presso il Dipartimento di Matematica e Informatica dell'Università di Parma, che supporta la programmazione a vincoli su insiemi all'interno del linguaggio Java. In JSetL è stato scelto di offrire come operazioni insiemistiche primitive soltanto uguaglianza ($=$), appartenenza ($\in$), unione ($\cup$) e disgiunzione ($\|$), mentre altre operazioni insiemistiche di base, quali inclusione ($\subseteq$), intersezione ($\cap$) e differenza ($\setminus$), sono introdotte come vincoli derivati.

L'obiettivo di questo lavoro di tesi è cambiare da derivata a primitiva l'implementazione dei vincoli insiemistici di inclusione (`subset`), intersezione (`inters`) e differenza (`diff`) nella libreria Java JSetL e valutare correttezza ed efficienza della nuova implementazione.

Dopo una breve presentazione della libreria JSetL, il lavoro sarà articolato in tre fasi: descrizione e implementazione delle regole di riscrittura (capitolo 2), valutazione dei vincoli implementati (capitolo 3) e trattamento dei casi speciali (capitolo 4).

Capitolo 1

JSetL

1.1 Insiemi

Gli insiemi in JSetL sono generalmente definiti istanziando la classe `LSet`. Per completezza diciamo che è anche possibile trattare insiemi tramite le classi `CP` (prodotto cartesiano tra insiemi), `Ris` (insiemi intensionali) e, in certi casi, `java.util.Set`; queste classi saranno però di interesse secondario in questo lavoro di tesi.

Distinguiamo principalmente cinque tipi di insiemi. Nei seguenti esempi si noti che il metodo `ins()` non modifica l'insieme di partenza, ma restituisce l'insieme risultato, che deve quindi essere assegnato ad una nuova variabile.

- **Empty set**: sono insiemi vuoti; in JSetL si ottengono nel seguente modo:

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

- **Unbound set**: sono insiemi non specificati; in JSetL questo termine è usato per indicare una variabile di tipo `LSet` che non ha ancora un valore assegnato (non inizializzata); ad esempio:

```
LSet A = new LSet("A");
```

  si noti che dato A unbound, risolvendo un'uguaglianza su A, come nel seguente modo:

```
Solver = new Solver();
solver.solve(A.eq(LSet.empty());
```

A diventa bound.

- **Ground set**: sono insiemi completamente specificati, ovvero insiemi del tipo $\{x_1, x_2, ..., x_n\}$ con x_i ground, ad esempio $\{1, 2, 3, 4, 5\}$. Questo tipo di insiemi in JSetL si ottiene aggiungendo elementi all'insieme vuoto:

```
LSet A = LSet.empty().ins(1,2,3,4,5);
```

dove A rappresenta l'insieme $\{1, 2, 3, 4, 5\}$. Si noti che ogni elemento x_i può anche essere una variabile logica legata ad un valore ground.

- **Non-ground bounded set**: sono insiemi parzialmente specificati del tipo $\{x_1, x_2, ..., x_n\}$, dove almeno uno degli x_i è una variabile logica non inizializzata; in JSetL questo tipo di insiemi si ottiene aggiungendo elementi all'insieme vuoto; ad esempio:

```
LVar x = new LVar("x"), y = new LVar("y");
LSet A = LSet.empty().ins(x,y);
```

dove A rappresenta l'insieme $\{x, y\}$. Si noti che la cardinalità di questo insieme può essere 1 o 2 a seconda che i valori assegnati a x e y siano uguali o diversi.

- **Non-ground unbounded set**: sono insiemi parzialmente specificati del tipo $\{x_1, x_2, ..., x_n/Y\}$, con x_i unbound o ground e Y variabile di tipo LSet non inizializzata; questo tipo di insiemi si ottiene aggiungendo elementi a un insieme unbound di partenza; ad esempio:

```
LVar x = new LVar("x");
LSet A = new LSet("B").ins(1,2,3,4,5,x);
```

dove A rappresenta l'insieme $\{1, 2, 3, 4, 5, x/B\}$.

Dato un insieme parzialmente specificato $X = \{x_1, x_2, ..., x_n/Y\}$, chiamiamo Y parte resto, o coda, di X. Inoltre si noti che gli unbounded set sono insiemi "aperti" (*open set*), nel senso che la loro cardinalità non è limitata a priori, mentre i ground set e i non ground bounded set sono insiemi "chiusi" (*closed set*), in quanto la loro cardinalità è compresa in limiti prefissati.

1.2 Vincoli primitivi e vincoli derivati

In JSetL le operazioni insiemistiche sono realizzate come vincoli. Ad esempio l'unione C tra due insiemi A e B, e cioè $C = A \cup B$, è realizzata dal vincolo `A.union(B,C)`, dove A è un oggetto di tipo `LSet`, e B e C sono oggetti di tipo `LSet` o `java.util.Set`.

Informalmente diciamo che un vincolo è **primitivo**, in JSetL, se per esso sono presenti regole di riscrittura ad-hoc, mentre è **derivato**, se viene gestito riscrivendolo in termini di altri vincoli primitivi.

In JSetL, tralasciano uguaglianza e appartenenza, le operazioni insiemistiche trattate come vincoli primitivi sono unione ($\cup$) e disgiunzione ($\|$), mentre inclusione ($\subseteq$), intersezione ($\cap$) e differenza ($\setminus$), sono trattate come vincoli derivati.

Questo è possibile perché date le operazioni di unione e disgiunzione, le seguenti regole permettono di scrivere, in funzione di esse, i vincoli di inclusione, intersezione e differenza:

$$s \subseteq t \Leftrightarrow \cup(s,t,t) \tag{1.1}$$

$$s \nsubseteq t \Leftrightarrow \neg \cup (s,t,t) \tag{1.2}$$

$$\cap (r,s,t) \Leftrightarrow \exists R,S(\cup(R,t,r) \wedge \cup(S,t,s) \wedge R \parallel S) \tag{1.3}$$

$$\neg \cap (r,s,t) \Leftrightarrow \exists(\cap(r,s,T) \wedge T \neq t) \tag{1.4}$$

$$\setminus (r,s,t) \Leftrightarrow \exists W(\cup(t,r,r) \wedge \cup(s,t,W) \wedge \cup(r,W,W) \wedge s \parallel t) \tag{1.5}$$

$$\neg \setminus (r,s,t) \Leftrightarrow \exists(\setminus(r,s,T) \wedge T \neq t) \tag{1.6}$$

Nella libreria JSetL i vincoli sono rappresentati dalla classe `Constraint`. Questa classe funge da contenitore di oggetti di tipo `AConstraint`. La classe `AConstraint` implementa i vincoli *atomici*, ovvero vincoli che non sono congiunzione di altri vincoli. Ogni oggetto istanza di questa classe contiene quattro campi `argument` per gli argomenti, un campo `solved` che indica quando il vincolo è in solved form e un attributo che contiene il codice identificativo del vincolo (`constraintKindCode`).

Come esempio di vincolo derivato, mostriamo la funzione che implementa il vincolo di intersezione in JSetL.

```
private void inters(@NotNull LSet lSet1, @NotNull LSet lSet2,
    @NotNull LSet lSet3, @NotNull AConstraint aConstraint) {
        assert lSet1 != null;
        assert lSet1.equ == null;
        assert lSet2 != null;
        assert lSet2.equ == null;
        assert lSet3 != null;
        assert lSet3.equ == null;
        assert aConstraint != null;
        assert aConstraint.constraintKindCode ==
            Environment.intersCode;
        assert aConstraint.argument1 != null;
        assert aConstraint.argument2 != null;
        assert aConstraint.argument3 != null;
        assert aConstraint.argument4 == null;

        LSet aux1 = new LSet();
        aConstraint.argument1 = aux1;
        aConstraint.argument2 = lSet3;
        aConstraint.argument3 = lSet1;
        aConstraint.argument4 = null;
        aConstraint.constraintKindCode = Environment.unionCode; //
            union(aux1,lSet3,lSet1)
        aConstraint.alternative = 0;

        LSet aux2 = new LSet();
        AConstraint c1 = new AConstraint(Environment.unionCode,
            aux2, lSet3,lSet2); // union(aux2,lSet3,lSet2)
        AConstraint c2 = new AConstraint(Environment.disjCode,
            aux1, aux2); // disj(aux1,aux2)
        solver.add(solver.indexOf(aConstraint) + 1, c1);
        solver.add(c2);
        solver.storeUnchanged = false;
}
```

Tralasciando i dettagli implementativi, che verranno illustrati nel capitolo 2, la funzione prende in input un generico vincolo `lSet1.inters(lSet2,lSet3)` e lo riscrive come congiunzione di tre vincoli primitivi: `aux1.union(lSet3,lSet1)`, `aux2.union(lSet3,lSet2)` e `aux1.disj(aux2)`.

1.3 Non determinismo

La libreria JSetL offre, tra le altre cose, strumenti per la gestione del non-determinismo. In particolare, sono due gli strumenti di interesse per questo lavoro di tesi.

La classe `Solver`, utilizzata per la risoluzione di congiunzioni di vincoli, offre il metodo `addChoicePoint(@NotNull AConstraint aConstraint)` che permette di creare un nuovo punto di scelta per il vincolo atomico passato come argomento.

La classe `AConstraint`, oltre a quelli già descritti, presenta il campo `alternative`, che identifica l'alternativa della risoluzione non-deterministica.

Vedremo nel capitolo 2 come utilizzare questi strumenti per implementare regole di risoluzione non-deterministiche per vincoli su insiemi.

Capitolo 2

Implementazione primitiva dei vincoli insiemistici subset, inters e diff

In questo capitolo descriviamo come trasformare i vincoli insiemistici di inclusione (`subset`), intersezione (`inters`) e differenza (`diff`), da derivati a primitivi, all'interno della libreria Java JSetL. Per fare questo è necessario individuare delle regole di riscrittura ad-hoc, che riportiamo nel sotto-capitolo 2.1. Nei sotto-capitoli 2.2 e 2.3 mostreremo poi l'implementazione concreta di queste regole all'interno di JSetL.

2.1 Regole di riscrittura astratte

Nelle seguenti regole, le variabili insiemistiche sono denotate con un punto sopra il nome della variabile (Ȧ); in tutti gli altri casi si intende un qualsiasi termine insiemistico.

2.1.1 Regole per subset $\subseteq$

$$\emptyset \subseteq A \rightarrow true \tag{2.1}$$

$$\dot{A} \subseteq \emptyset \rightarrow \dot{A} = \emptyset \tag{2.2}$$

$$\dot{A} \subseteq \{x_1, ..., x_n | B\} \rightarrow irreducible \tag{2.3}$$

$$\{x_1, ..., x_n | A\} \subseteq \emptyset \rightarrow false \tag{2.4}$$

$$\{x \sqcup A\} \subseteq \dot{B} \rightarrow \dot{B} = \{x \sqcup \dot{N}\} \wedge A \subseteq \{x \sqcup \dot{N}\} \tag{2.5}$$

$$\begin{gathered} \{x \sqcup A\} \subseteq \{y \sqcup B\} \rightarrow x = y \wedge A \subseteq \{y \sqcup B\} \\ \bigvee \\ x \neq y \wedge x \in B \wedge A \subseteq \{y \sqcup B\} \end{gathered} \tag{2.6}$$

$$\emptyset \subseteq [m, n] \rightarrow true \tag{2.8}$$

$$\dot{A} \subseteq [m, n] \rightarrow irreducible \tag{2.9}$$

$$\{x \sqcup A\} \subseteq [m, n] \rightarrow m \leq x \leq n \wedge A \subseteq [m, n] \tag{2.10}$$

Nella nostra implementazione scegliamo di avere il minor numero di vincoli irriducibili, quindi la regola 3 è sostituita con la seguente regola 7:

$$\dot{A} \subseteq \{x \sqcup B\} \rightarrow A = \{x \sqcup N\} \wedge x \notin N \wedge N \subseteq B \bigvee x \notin A \wedge A \subseteq B \tag{2.7}$$

In questo modo l'unico caso irriducibile è $\dot{A} \subseteq \dot{B}$.

2.1.2 Regole per inters ∩

$$inters(\dot{A}, t, \dot{C}) \rightarrow irreducible \tag{2.11}$$

$$inters(t, \dot{B}, \dot{C}) \rightarrow irreducible \tag{2.12}$$

$$inters(\emptyset, B, C) \rightarrow C = \emptyset \tag{2.13}$$

$$inters(A, \emptyset, C) \rightarrow C = \emptyset \tag{2.14}$$

$$inters(A, B, \emptyset) \rightarrow A \parallel B \tag{2.15}$$

$$\begin{gathered} inters(A, B, \{x \sqcup C\}) \rightarrow \\ A = \{x \sqcup N_1\} \wedge B = \{x \sqcup N_2\} \wedge inters(N_1, N_2, C) \end{gathered} \tag{2.16}$$

Per eliminare i casi irriducibili (regole 11 e 12) si introduce la seguente regola:

$$\begin{aligned} inters(A, \{x \sqcup B\}, \dot{C}) \rightarrow \\ A = \{x \sqcup N\} \wedge x \notin N \wedge C = \{x \sqcup N_1\} \wedge x \notin N_1 \wedge inters(N, B, N_1) \\ \bigvee \\ x \notin A \wedge inters(A, B, C) \end{aligned} \tag{2.18}$$

La regola per il caso $inters(\{x|A\}, B, C)$ è simmetrica alla regola 18.

$$\begin{aligned} inters(\{x \sqcup A\}, \dot{B}, \dot{C}) \rightarrow \\ B = \{x \sqcup N\} \wedge x \notin N \wedge C = \{x \sqcup N_1\} \wedge x \notin N_1 \wedge inters(A, N, N_1) \\ \bigvee \\ x \notin B \wedge inters(A, B, C) \end{aligned} \tag{2.18 bis}$$

In questo modo l'unico caso irriducibile è $inters(\dot{A}, \dot{B}, \dot{C})$.

2.1.3 Regole per diff $\setminus$

$$diff(\dot{A}, \dot{B}, \dot{C}) \rightarrow irreducible \tag{2.19}$$

$$diff(\emptyset, B, C) \rightarrow C = \emptyset \tag{2.20}$$

$$diff(A, \emptyset, C) \rightarrow C = A \tag{2.21}$$

$$diff(A, B, \emptyset) \rightarrow A \subseteq B \tag{2.22}$$

$$diff(A, B, \{x \sqcup C\}) \rightarrow A = \{x \sqcup N\} \wedge x \notin N \wedge x \notin B \wedge diff(N, B, C) \tag{2.23}$$

$$\begin{aligned} diff(A, \{x \sqcup B\}, C) \rightarrow A = \{x \cup N\} \wedge x \notin N \wedge diff(N, B, C) \\ \bigvee \\ x \notin A \wedge diff(A, B, C) \end{aligned} \tag{2.24}$$

$$\begin{aligned} diff\{x \sqcup A\}, B, C) \rightarrow x \notin B \wedge C = \{x \sqcup N\} \wedge diff(A, B, N) \\ \bigvee \\ B = \{x \sqcup N\} \wedge x \notin N \wedge diff(A, N, C) \end{aligned} \tag{2.25}$$

2.2 Scelte implementative generali

Per implementare le nuove regole scegliamo di estendere `RwRulesSet`, la classe di JSetL contenente le regole di riscrittura per vincoli insiemistici (ad eccezione dei vincoli di uguaglianza e disuguaglianza) creando la classe `RwRulesSetExtd`. Questa scelta permette di concentrarsi solo sui vincoli di interesse (in questo caso `subset`, `inters` e `diff`), col vantaggio di non dover gestire codice inutile ai fini del nostro lavoro; si noti infatti che tutti gli altri vincoli continueranno ad essere gestiti dalla vecchia implementazione (RwRulesSet). Il secondo vantaggio è la trasparenza rispetto ad eventuali aggiornamenti della classe durante lo svolgimento del lavoro.

Per applicare questa tecnica sono necessari i seguenti accorgimenti:

- `RwRulesSetExtd` deve definire come campi private gli handler per vincoli di uguaglianza (`eqHandler`) e per vincoli su insiemi finiti (`FSHandler`). Questo accorgimento è necessario in quanto in `RwRulesSet`, questi sono definiti come campi private e quindi non accessibili dalla sottoclasse. Un'altra soluzione sarebbe stata ridefinire come protected tali campi, ma l'obiettivo della scelta implementativa che stiamo descrivendo è non modificare la classe base preesistente.
- Il costruttore di `RwRulesSetExtd`, oltre a inizializzare i campi, deve invocare il costruttore della classe madre.

```
public class RwRulesSetExtd extends RwRulesSet{
    /**
      * Rewrite rules used to handle equality constraints.
      */
    private RwRulesEq eqHandler;

    /**
      * Rewrite rules used to handle constraints over {@code
          SetLVar}s.
      */
    private RwRulesFS FSHandler;


    public RwRulesSetExtd(@NotNull Solver solver) {
      super(solver);

      eqHandler = new RwRulesEq(solver);
```

```
        FSHandler = new RwRulesFS(solver);
    }

    //Implementation...
}
```

- É necessario modificare il costruttore della classe `Solver`, in modo che nell'array degli handler di vincoli venga inserito un oggetto di tipo `RwRulesSetExtd` e non più `RwRulesSet` (unica modifica alle classi preesistenti).

```
public Solver() {
    constraintHandlers[nHandlers++] = new RwRulesEq(this);

    constraintHandlers[nHandlers++] = new RwRulesFD(this);

    //constraintHandlers[nHandlers++] = new RwRulesSet(this);
    constraintHandlers[nHandlers++] = new RwRulesSetExtd(this);

    constraintHandlers[nHandlers++] = new RwRulesFS(this);

    constraintHandlers[nHandlers++] = new RwRulesMeta(this);

    constraintHandlers[nHandlers++] = new RwRulesBool(this);

    constraintHandlers[nHandlers++] = new RwRulesBR(this);

    neqRemover = new NeqRemover(this);

}
```

2.3 Implementazione delle regole di riscrittura

Assumiamo di voler trattare un generico vincolo *nomeVincolo*. In JSetL, i vincoli insiemistici sono gestiti nel seguente modo.

- Una prima funzione `nomeVincolo` che prende in input un oggetto `aConstraint` di tipo `AConstraint`, ne controlla gli argomenti e in base ai tipi degli argomenti chiama le apposite funzioni; più precisamente:
  - se uno o più argomenti sono di tipo `Set`, li incapsula in oggetti di tipo `LSet` e richiama se stessa sullo stesso `aConstraint`;
  - se uno o più argomenti sono di tipo `Ris` o `CP`, chiama le funzioni apposite;
  - se l'argomento risultante è di tipo `SetLVar`, passa il vincolo all'handler per insiemi finiti;
  - se tutti gli argomenti sono di tipo `LSet`, chiama la funzione `nomeVincoloRule` che prende in input gli argomenti del vincolo e il vincolo stesso, e gestisce i vari casi presentati nelle regole astratte.
- I casi possono essere gestiti direttamente dalla funzione `nomeVincoloRule` oppure tale funzione può chiamare in una sequenza di if-else esclusivi le funzioni `nomeVincoloRuleNumero_regola` che implementano singolarmente le varie regole; questo secondo metodo è quello da noi scelto, in quanto favorisce maneggiabilità, leggibilità e riuso del codice.
- Ogni singola funzione `nomeVincoloRuleNumero_regola` verifica se il vincolo ricade nel caso di propria competenza; in caso negativo ritorna false, altrimenti prova a risolverlo e, se ha successo, ritorna true.

Si noti che all'interno della libreria i nomi delle funzioni variano leggermente; quella appena descritta è la convenzione da noi adottata per questo lavoro di tesi.

2.3.1 Implementazione del vincolo di inclusione

Il metodo `subset` è il corrispondente di `nomeVincolo` per il vincolo di inclusione. Ne riportiamo di seguito l'implementazione.

```
@Override
protected void subset(@NotNull AConstraint aConstraint) {
    assert aConstraint != null;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null || aConstraint.argument4
        instanceof LSet;
```

```
    assert aConstraint.constraintKindCode == Environment.subsetCode;

    manageEquChains(aConstraint);

    if (aConstraint.argument1 instanceof SetLVar) // setlvar subset
        lset
      FSHandler.subset(aConstraint);

    else if (aConstraint.argument1 instanceof Ris ||
        aConstraint.argument2 instanceof Ris) //Ris subset Ris
      subsetDerived((LSet) aConstraint.argument1, (LSet)
          aConstraint.argument2, aConstraint);

    else if (aConstraint.argument1 instanceof CP &&
        aConstraint.argument2 instanceof LSet) // cp subset lset,
        cp subset cp
      subsetDerived((CP) aConstraint.argument1, (LSet)
          aConstraint.argument2, aConstraint);

    else if (aConstraint.argument1 instanceof LSet &&
        aConstraint.argument2 instanceof CP) // lset subset cp
      subsetDerived((LSet) aConstraint.argument1, (CP)
          aConstraint.argument2, aConstraint);

    else if (aConstraint.argument1 instanceof LSet &&
        aConstraint.argument2 instanceof MultiInterval)
      subsetRuleInterval((LSet)aConstraint.argument1,
          (MultiInterval)aConstraint.argument2, aConstraint);

    else if (aConstraint.argument1 instanceof LSet &&
        aConstraint.argument2 instanceof LSet) // lset subset lset
      subsetRule((LSet) aConstraint.argument1, (LSet)
          aConstraint.argument2, aConstraint);

    else if (aConstraint.argument1 instanceof LSet &&
        aConstraint.argument2 instanceof Set) { // lset subset set
      LSet aux = new LSet((Set<?>)aConstraint.argument2);
      aConstraint.argument2 = aux;
      subset(aConstraint);
    }

    return;
}
```

Il metodo `subsetDerived` gestisce tutti i casi in cui gli argomenti non sono di tipo `LSet` o `Set`, nello stesso modo in cui avveniva nella vecchia implementazione; più precisamente, nel caso di argomenti di tipo `CP`, controlla se sono espandibili e in caso affermativo li espande; poi riscrive il vincolo in termini di unione. Qui potrebbe essere utile inserire un'ottimizzazione analoga nel caso di argomenti di tipo `Ris`.

```
private void subsetDerived(@NotNull LSet lSet1, @NotNull LSet
    lSet2, @NotNull AConstraint aConstraint) {
    assert lSet1 != null;
    assert lSet1.equ == null;
    assert lSet2 != null;
    assert lSet2.equ == null;
    assert aConstraint != null;
    assert aConstraint.constraintKindCode == Environment.subsetCode;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null;

    if(lSet1.isBound() && lSet1.isGround() && lSet1 instanceof CP)
      lSet1 = ((CP) lSet1).expand();
    if(lSet2.isBound() && lSet2.isGround() && lSet2 instanceof CP)
      lSet2 = ((CP) lSet2).expand();

    aConstraint.argument1 = lSet1;
    aConstraint.argument2 = lSet2;
    aConstraint.argument3 = lSet2;
    aConstraint.argument4 = null;
    aConstraint.constraintKindCode = Environment.unionCode;
    aConstraint.alternative = 0;
    solver.storeUnchanged = false;
}
```

Il metodo `subsetRule` chiama in sequenza le funzioni che implementano le singole regole ad-hoc: se la regola chiamata ha successo, ritorna, altrimenti procede nella sequenza di chiamate.

```
private void subsetRule(@NotNull LSet lSet1, @NotNull LSet lSet2,
    @NotNull AConstraint aConstraint) {
    assert lSet1 != null;
```

```
    assert lSet1.equ == null;
    assert lSet2 != null;
    assert lSet2.equ == null;
    assert aConstraint != null;
    assert aConstraint.constraintKindCode == Environment.subsetCode;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null;

    if (subsetRule1(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule2(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule3_7(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule4(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule5(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule6(lSet1, lSet2, aConstraint)) return;
    else return;

}
```

Si noti come la funzione `subsetRuleInterval`, che implementa le regole astratte sugli intervalli (2.8, 2.9, 2.10), non venga chiamata in `subsetRule`, ma in `subset`; questa scelta è dovuta al fatto che la verifica dei tipi degli argomenti del vincolo, è effettuate nella funzione `subset`.

Procediamo ora ad illustrare il procedimento per passare da una generica regola astratta N alla corrispondente funzione `nomeVincoloRuleN` che la implementa. Per farlo prendiamo come esempio la regola 2.7

$$\dot{A} \subseteq \{x \sqcup B\} \rightarrow A = \{x \sqcup N\} \wedge x \notin N \wedge N \subseteq B \bigvee x \notin A \wedge A \subseteq B \quad (2.7)$$

e implementiamo la corrispondente funzione `subsetRule3_7` (è indicata in questo modo in quanto la regola 7 è la riscrittura, volta a eliminare l'irriducibilità, della regola 3).

1. I parametri della funzione devono essere gli insiemi coinvolti nel vincolo e il vincolo stesso; quindi nel nostro caso la testa della funzione sarà la seguente.

```
private boolean subsetRule3_7(@NotNull LSet lSet1, @NotNull
    LSet lSet2, @NotNull AConstraint aConstraint)
```

2. La funzione deve controllare che:

   - gli insiemi non siano nulli;
   - gli insiemi non siano legati ad altri insiemi da vincoli di uguaglianza o, se lo sono, siano gli ultimi a destra nella catena di uguaglianze; questo è fatto controllando che il campo `equ` dell'oggetto `LSet` sia nullo;
   - il vincolo non sia nullo;
   - il codice del vincolo corrisponda al tipo di vincolo che la funzione implementa;
   - non siano nulli tutti e soli gli argomenti del vincolo necessari; come già detto, un oggetto di tipo `AConstraint` ha sempre quattro campi argomento; ad esempio nel caso di un vincolo di inclusione, che coinvolge due soli insiemi, serve quindi verificare che non siano nulli i primi due argomenti e che lo siano i secondi due.

   Così facendo la funzione è la seguente:

```
private boolean subsetRule3_7(@NotNull LSet lSet1, @NotNull
    LSet lSet2, @NotNull AConstraint aConstraint) {
    assert lSet1 != null;
    assert lSet1.equ == null;
    assert lSet2 != null;
    assert lSet2.equ == null;
    assert aConstraint != null;
    assert aConstraint.constraintKindCode ==
        Environment.subsetCode;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null;

        //...
}
```

3. La funzione `nomeVincoloRule` chiama in sequenza le funzioni che implementano le regole, ritornando dopo aver trovato la funzione che risolve il caso desiderato. Per fare questo è necessario che ogni funzione `nomeVincoloRuleNumero_regola` controlli se il vincolo rientra nella propria casistica:

- se il controllo ha esito positivo, riscrive il vincolo e ritorna true;
- altrimenti ritorna immediatamente false.

Nel nostro caso, la regola 2.7 prevede che A sia unbound e B contenga almeno un elemento. Il codice sarà quindi il seguente:

```
  private boolean subsetRule3_7(@NotNull LSet lSet1, @NotNull
      LSet lSet2, @NotNull AConstraint aConstraint) {
    assert lSet1 != null;
    assert lSet1.equ == null;
    assert lSet2 != null;
    assert lSet2.equ == null;
    assert aConstraint != null;
    assert aConstraint.constraintKindCode ==
        Environment.subsetCode;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null;

    if (!lSet1.isBound() && lSet2.isBound()) {
      //implementazione

      return true;
    }

    return false;
}
```

Il metodo di `LSet isBound()` ritorna true se l'insieme è bound, false altrimenti.

4. La regola 2.7 presenta un or, prevede cioè due possibili casi di riscrittura ed è quindi una regola non-deterministica. Per implementarla utilizziamo gli strumenti illustrati nel sottocapitolo 2.3, nel seguente modo: la funzione effettua un controllo sull'alternativa non-deterministica in esecuzione, tramite `switch(aConstraint.alternative)` e aggiunge un punto di scelta in tutti i rami tranne l'ultimo.

```
private boolean subsetRule3_7(@NotNull LSet lSet1, @NotNull
    LSet lSet2, @NotNull AConstraint aConstraint) {
```

```
    assert lSet1 != null;
    assert lSet1.equ == null;
    assert lSet2 != null;
    assert lSet2.equ == null;
    assert aConstraint != null;
    assert aConstraint.constraintKindCode ==
        Environment.subsetCode;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null;

    if (!lSet1.isBound() && lSet2.isBound()) {
      switch (aConstraint.alternative) {
        case 0:
          solver.addChoicePoint(aConstraint);

          //implementazione riscrittura 1

          aConstraint.alternative = 0;

          return true;

        case 1:
          //implementazione riscrittura 2

          aConstraint.alternative = 0;

          return true;
      }
    }

    return false;
}
```

5. Infine traduciamo le regole di riscrittura, con le seguenti linee guida:

   - è sempre conveniente modificare il vincolo passato in input, prima di crearne di nuovi; se si modifica il vincolo invece che crearne uno nuovo allora bisogna inizializzare a 0 il campo `alternative`, in modo che eventuali scelte non-deterministiche nel vincolo modificato partano dalla prima;

- un nuovo vincolo atomico è creato tramite il costruttore della classe `AConstraint`, passando il codice identificativo del vincolo e gli argomenti, in questo ordine;
- i vincoli sono aggiunti allo store del solver tramite il metodo `add()`;
- l'attributo `storeUnchanged` del solver serve a segnalare se il contenuto del solver stesso è rimasto invariato, va quindi posto a false ogni volta che si modifica il vincolo in ingresso e/o si aggiungono al solver altri vincoli.

Il codice che otteniamo è il seguente.

```
private boolean subsetRule3_7(@NotNull LSet lSet1, @NotNull
    LSet lSet2, @NotNull AConstraint aConstraint) {
    assert lSet1 != null;
    assert lSet1.equ == null;
    assert lSet2 != null;
    assert lSet2.equ == null;
    assert aConstraint != null;
    assert aConstraint.constraintKindCode ==
        Environment.subsetCode;
    assert aConstraint.argument1 != null;
    assert aConstraint.argument2 != null;
    assert aConstraint.argument3 == null;
    assert aConstraint.argument4 == null;

    if (!lSet1.isBound() && lSet2.isBound()) {
      switch (aConstraint.alternative) {
        case 0:
          solver.addChoicePoint(aConstraint);

          LSet N = new LSet();

          AConstraint s1 = new AConstraint(Environment.eqCode,
              lSet1, N.ins(lSet2.getOne()));
          AConstraint s2 = new
              AConstraint(Environment.ninCode, lSet2.getOne(),
              N);

          solver.add(s1);
          solver.add(s2);

          aConstraint.argument1 = N;
```

```
            aConstraint.argument2 = lSet2.removeOne();
            aConstraint.alternative = 0;

            solver.storeUnchanged = false;

            return true;

          case 1:

            AConstraint s3 = new
                AConstraint(Environment.ninCode, lSet2.getOne(),
                lSet1);
            solver.add(s3);

            aConstraint.argument2 = lSet2.removeOne();
            aConstraint.alternative = 0;

            solver.storeUnchanged = false;

            return true;
        }
      }

      return false;
    }
```

Si è scelto di implementare la regola 2.7 perché è la più complessa; analoghe sono le implementazioni delle altre regole di riscrittura per il vincolo di inclusione. Come caso di interesse riportiamo di seguito solo l'implementazione del metodo `subsetRuleInterval`.

```
private void subsetRuleInterval(@NotNull LSet lSet, @NotNull
    MultiInterval interval, @NotNull AConstraint aConstraint) {
    /*
     * assert varie
     */

    if (lSet.isBoundAndEmpty()) { // rule 8: {} subset [m,n]
      aConstraint.setSolved(true);

      return;
```

```
    }

    else if (!lSet.isBound()) { // rule 9: A subset [m,n]
      return;
    }

    else if (lSet.isBound()) { // rule 10: {x|A} subset [m,n]

      AConstraint s;
      Object setElem = lSet.getOne();

      if(setElem instanceof Integer) {
        s = new AConstraint(Environment.inCode, new
            IntLVar((Integer)setElem), interval);

      } else if(setElem instanceof LVar){
        s = new AConstraint(Environment.inCode, (LVar)setElem,
            interval);

      } else {
        solver.fail(aConstraint);
        return;
      }

      solver.add(s);

      aConstraint.argument1 = lSet.removeOne();

      solver.storeUnchanged = false;
    }
}
```

Si noti che nel caso di un vincolo che implica il vero si dichiara che il vincolo è risolto, tramite il metodo `setSolved()` e si ritorna; mentre in caso di vincolo irriducibile ci si limita a ritornare.

Sempre nell'idea che la riscrittura miri all'efficienza, decidiamo di aggiungere una regola che non era possibile introdurre in astratto, perché legata all'implementazione: quando i due argomenti del vincolo di inclusione sono lo stesso reference ad un insieme unbound, si conclude che il vincolo è risolto.

$$\dot{A} \subseteq \dot{A} \rightarrow \mathit{true} \qquad \text{(subsetRuleExtra)}$$

Banalmente l'implementazione della regola `subsetRuleExtra` sarà la seguente.

```
private boolean subsetRuleExtra(@NotNull LSet lSet1, @NotNull LSet
    lSet2, @NotNull AConstraint aConstraint) {
    /*
    * assert varie
    */

    if((!lSet1.isBound() && !lSet2.isBound() && lSet1 == lSet2)) {
      aConstraint.setSolved(true);

      return true;
    }

    return false;
}
```

La nuova funzione `subsetRule` sarà quindi la seguente.

```
private void subsetRule(@NotNull LSet lSet1, @NotNull LSet lSet2,
    @NotNull AConstraint aConstraint) {
    /*
     * assert varie
     */

    if (subsetRuleExtra(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule1(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule2(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule3_7(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule4(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule5(lSet1, lSet2, aConstraint)) return;
    else if (subsetRule6(lSet1, lSet2, aConstraint)) return;
    else return;

}
```

Analoghe sono le regole per intersezione e differenza.

$$inters(\dot{A}, \dot{A}, B) \rightarrow B = \dot{A} \qquad \text{(intersRuleExtra)}$$

$$diff(\dot{A}, \dot{A}, B) \rightarrow B = \emptyset \qquad \text{(diffRuleExtra)}$$

Piú in generale le implementazioni come vincoli primitivi di intersezione e differenza sono del tutto analoghe a quella di inclusione appena descritta.

Capitolo 3

Valutazione dei vincoli implementati

Dopo aver implementato le regole, procediamo valutandone correttezza ed efficienza.

3.1 Valutazione di correttezza

Questa tesi, per vari motivi, non si pone l'obiettivo di effettuare un testing completo delle regole implementate; ci limiteremo ad individuare una casistica ragionevolmente ampia, che permetta di valutare i casi più comuni.

Più precisamente, procediamo a valutare ogni regola astratta, eventualmente individuando più casi all'interno di essa. In linea di principio si prendono in considerazione 5 tipi di insieme: *empty set*, *unbound set*, *ground set*, *closed non-ground set* e *open non-ground set*.

Una volta individuati i casi, quello che faremo sarà verificare che le soluzioni generate, spesso tutte, ma in certi casi le prime N (tipicamente $N = 100$), rispettino le caratteristiche attese, nei seguenti modi:

- controllando il contenuto dello store del solver, tramite il metodo `solver.getConstraint()`;
- effettuando controlli mirati sulle parti specificate degli insiemi; ad esempio, in astratto, dato un vincolo del tipo $inters(\{1, x/A\}, \{2, 3, x/B\}, C)$ è evidente la conseguenza $x \in C$;
- risolvendo altri vincoli sugli insiemi coinvolti; ad esempio data la regola 2.15: $inters(A, B, \emptyset) \rightarrow A \parallel B$, verifichiamo che A e B siano effettivamente disgiunti, risolvendo il vincolo `A.disj(B)` (indicheremo come

vincoli di "test" questi vincoli aggiuntivi usati esclusivamente per il testing).

Per la risoluzione dei vincoli di "test", usare il solver (S) utilizzato nella risoluzione dei vincoli testati potrebbe modificare il contenuto dello store, causando un comportamento diverso da quello che si avrebbe in un'esecuzione standard.

Nel caso serva è quindi conveniente definire un solver ausiliario (S_{aux}). L'inconveniente di questa soluzione è che S_{aux} non conosce il contenuto dello store di S. Per ovviare a questo problema semplicemente aggiungiamo a S_{aux} la congiunzione di vincoli contenuta nello store di S.

```
Solver s_aux = new Solver();
s_aux.add(s.getConstraint());
```

Riportiamo ora i casi concreti scelti per la valutazione. Quando scriviamo le prime lettere dell'alfabeto in maiuscolo (A, B, C, ...) intendiamo insiemi unbound, mentre con le lettere minuscole x, y, z indichiamo `LVar` unbound; inoltre indichiamo eventualmente con le prime lettere dell'alfabeto minuscole a, b, c variabili intere ground (`IntLVar`); infine PS sta per Partially Specified (ricordiamo che insieme parzialmente specificato è sinonimo di insieme non-ground). Solo per i casi semplici è indicato come dovrebbe concludersi la risoluzione.

3.1.1 Test del vincolo subset

Presentiamo i casi testati per la regola di inclusione.

lSet1 \ lSet2	Unbound
Unbound	$subset(A, A) \rightarrow true$

Tabella 3.1: Rule extra

lSet1 \ lSet2	Unbound	Ground	Open PS
$\emptyset$	$subset(\emptyset, A) \rightarrow true$	$subset(\emptyset, \{1, 10\}) \rightarrow true$	$subset(\emptyset, \{1/A\}) \rightarrow true$

Tabella 3.2: Rule 1

lSet1 \ lSet2	$\emptyset$
Unbound	$subset(A, \emptyset) \rightarrow A = \emptyset$

Tabella 3.3: Rule 2

lSet1 \ lSet2	Ground	Closed PS	Open PS	Open PS 2
Unbound	$subset(A, \{1\})$	$subset(A, \{x\})$	$subset(A, \{x/B\})$	$subset(A, \{1, x/B\})$

Tabella 3.4: Rule 3-7

lSet1 \ lSet2	$\emptyset$
Ground	$subset(\{1\}, \emptyset) \rightarrow false$
Closed PS	$subset(\{x\}, \emptyset) \rightarrow false$
Open PS	$subset(\{x/A\}, \emptyset) \rightarrow false$
Open PS 2	$subset(\{1, x/A\}, \emptyset) \rightarrow false$

Tabella 3.5: Rule 4

lSet1 \ lSet2	Unbound
Ground	$subset(\{1\}, B)$
Closed PS	$subset(\{x\}, B)$
Open PS	$subset(\{x/A\}, B)$
Open PS 2	$subset(\{1, x/A\}, B)$

Tabella 3.6: Rule 5

lSet1 \ lSet2	Ground	Closed PS	Open PS
Ground	$subset(\{2\}, \{2\})$	$subset(\{2\}, \{y\})$	$subset(\{2\}, \{y/B\})$
Closed PS	$subset(\{x\}, \{2\})$	$subset(\{x\}, \{y\})$	$subset(\{x\}, \{y/B\})$
Open PS	$subset(\{x/A\}, \{2\})$	$subset(\{x/A\}, \{y\})$	$subset(\{x/A\}, \{y/B\})$
Open PS 2	$subset(\{2, x/A\}, \{2\})$	$subset(\{2, x/A\}, \{y\})$	$subset(\{2, x/A\}, \{y/B\})$

Tabella 3.7: Rule 6

lSet1 \ lSet2	Interval
Unbound	$subset(A, [10..3]) \rightarrow A = \emptyset$
$\emptyset$	$subset(\emptyset, [1..10]) \rightarrow true$
Unbound	$subset(A, [1..10]) \rightarrow irreducible$
Ground	$subset(\{1\}, [1..10])$
Closed PS	$subset(\{x\}, [1..10])$
Open PS	$subset(\{x/A\}, [1..10])$
Open PS 2	$subset(\{1, x, /A\}, [1..10])$

Tabella 3.8: Rule Interval

Riportiamo come esempi i programmi per la regola 1 e per il caso particolare in cui il secondo argomento del vincolo sia un intervallo.

```
@Test
public void testSubsetRule1() throws Failure {
    Solver solver = new Solver();

    LSet S1 = LSet.empty().setName("S1");

    //unbound lSet2
    LSet S2 = new LSet("S2");

    assertTrue(solver.check(S1.subset(S2)));

    assertEquals("[]", solver.getConstraint().toString());

    //ground lSet2
    S2 = LSet.empty().ins(1,10).setName("S2");

    assertTrue(solver.check(S1.subset(S2)));

    assertEquals("[]", solver.getConstraint().toString());

    //partially bound lSet2
    LSet A = new LSet("A");
    LVar x = new LVar(1);
    S2 = A.ins(x).setName("S2");

    assertTrue(solver.check(S1.subset(S2)));
```

```
    assertEquals("[]", solver.getConstraint().toString());

}
```

```
@Test
public void testSubsetRuleInterval() throws Failure {
    Solver solver = new Solver();

    MultiInterval S2 = new MultiInterval(10,3);

    //LSet subset [a..b], con a>b --> LSet = {}
    LSet S1 = new LSet("S1");

    solver.solve(S1.subset(S2));

    assertEquals("subset(_S1,{})",
        solver.getConstraint().toString());
    assertFalse(solver.nextSolution());

    //{} subset [a..b] (a =< b) --> true
    S1 = LSet.empty();
    S2 = new MultiInterval(1,10);

    solver.clearStore();
    solver.solve(S1.subset(S2)); //rule 8

    assertEquals("[]", solver.getConstraint().toString());
    assertFalse(solver.nextSolution());

    //A subset [a..b] (unbound A, a =< b) --> irreducible
    S1 = new LSet("S1");

    solver.solve(S1.subset(S2));

    assertEquals("subset(_S1,[1..10])",
        solver.getConstraint().toString());
    assertFalse(solver.nextSolution());


    //{x/A} subset [a..b] (a =< b)
    //{x|A}: //ground lset: ground x, A = {} (es. {1})
```

```
S1 = LSet.empty().ins(1).setName("S1");

solver.clearStore();
solver.solve(S1.subset(S2));
assertEquals("[]", solver.getConstraint().toString());
assertFalse(solver.nextSolution());

//partially specified (closed) lset: unbound x, A = {} (es. {x})
LVar x = new LVar("x");
S1 = LSet.empty().ins(x).setName("S1");

solver.solve(S1.subset(S2));

do {  //x get all values in [1,10]
  assertTrue(solver.check(x.in(S2)));
  assertEquals("[]", solver.getConstraint().toString());
} while(solver.nextSolution());


//partially specified (open) lset: unbound x, unbound A
x = new LVar("x");
S1 = new LSet("A").ins(x).setName("S1");

solver.solve(S1.subset(S2));

do {  //x get all values in [1,10]
  assertTrue(solver.check(x.in(S2)));
  assertEquals("subset(_A,[1..10])",
      solver.getConstraint().toString());
} while(solver.nextSolution());


//more elements: es. {1,x|A}
x = new LVar("x");
S1 = new LSet("A").ins(1,x).setName("S1");

solver.clearStore();
solver.solve(S1.subset(S2));

do {  //x get all values in [1,10]
  assertTrue(solver.check(x.in(S2)));
  assertEquals("subset(_A,[1..10])",
      solver.getConstraint().toString());
```

```
    } while(solver.nextSolution());

}
```

Dove possibile è poi necessario verificare la correttezza anche nei corrispondenti casi false, ovvero i casi la cui risoluzione dovrebbe terminare con un fallimento. Di seguito riportiamo il programma di valutazione della regola 4.

```
@Test
public void testSubsetRule4() throws Failure {
    Solver solver = new Solver();

    //ground lset1: ground x, A = {} (es. {1})
    LSet S1 = LSet.empty().ins(1).setName("S1");
    LSet S2 = LSet.empty().setName("S2");

    assertFalse(solver.check(S1.subset(S2)));

    //partially specified (closed) lset1: unbound x, A = {} (es.
       {x})
    LVar x = new LVar("x");
    S1 = LSet.empty().ins(x).setName("S1");

    solver.clearStore();
    assertFalse(solver.check(S1.subset(S2)));

    //partially specified (open) lset1: unbound x, unbound A
    S1 = new LSet("A").ins(x);

    solver.clearStore();
    assertFalse(solver.check(S1.subset(S2)));

    //more elements lSet1: es. {1,x|A}
    S1 = new LSet("A").ins(1,x);

    solver.clearStore();
    assertFalse(solver.check(S1.subset(S2)));
}
```

Si noti che sottoponendo vincoli la cui risoluzione dovrebbe fallire, perchè

non soddisfacibili, si utilizza la funzione `assertFalse()` e il metodo `check()` al posto di `solve()`, per comodità di esecuzione dei test junit.

3.1.2 Test del vincolo inters

Presentiamo i casi di test della regola di intersezione.

lSet3	lSet1 \ lSet2	Unbound
Unbound	Unbound	$inters(A, A, C)$

Tabella 3.9: Rule extra

lSet3	lSet1 \ lSet2	Unbound	Ground	Open PS
Unbound	$\emptyset$	$inters(\emptyset, B, C)$	$inters(\emptyset, \{1, 2\}, C)$	$inters(\emptyset, \{x, y/B\}, C)$

Tabella 3.10: Rule 13

lSet3	lSet1 \ lSet2	$\emptyset$
Unbound	Unbound	$inters(A, \{\emptyset\}, C)$
	Ground	$inters(\{1, 2\}, \{\emptyset\}, C)$
	Open PS	$inters(\{x, y/A\}, \{\emptyset\}, C)$

Tabella 3.11: Rule 14

lSet3	lSet1 \ lSet2	Unbound	Ground
$\emptyset$	Unbound	$inters(A, B, \emptyset)$	$inters(A, \{1, 2\}, \emptyset)$
	Ground	$inters(\{3, 4\}, B, \emptyset)$	$inters(\{3, 4\}, \{1, 2\}, \emptyset)$
	Open PS	$inters(\{w, z/A\}, B, \emptyset)$	$inters(\{w, z/A\}, \{1, 2\}, \emptyset)$

lSet3	lSet1 \ lSet2	Open PS
$\emptyset$	Unbound	$inters(A, \{x, y/B\}, \emptyset)$
	Ground	$inters(\{3, 4\}, \{x, y/B\}, \emptyset)$
	Open PS	$inters(\{w, z/A\}, \{x, y/B\}, \emptyset)$

Tabella 3.12: Rule 15

lSet3	lSet2 / lSet1	Unbound	Ground
Ground	Unbound	$inters(A, B, \{1,2\})$	$inters(A, \{1,2,3,4\}, \{1,2\})$
	Ground	$inters(\{1,2,5,6\}, B, \{1,2\})$	$inters(\{1,2,5,6\}, \{1,2,3,4\}, \{1,2\})$
	Open PS	$inters(\{w, z/A\}, B, \{1,2\})$	$inters(\{w, z/A\}, \{1,2,5,6\}, \{1,2\})$

lSet3	lSet2 / lSet1	Open PS
Ground	Unbound	$inters(A, \{x, y/B\}, \{1,2\})$
	Ground	$inters(\{1,2,5,6\}, \{x, y/B\}, \{1,2\})$
	Open PS	$inters(\{w, z/A\}, \{x, y/B\}, \{1,2\})$

Tabella 3.13: Rule 16-18 with lSet3 ground

lSet3	lSet2 / lSet1	Unbound	Ground
Unbound	Unbound	$inters(A, B, C)$	$inters(A, \{1,2\}, C)$
	Ground	$inters(\{1,2\}, B, C)$	$inters(\{1,2\}, \{1,2,3\}, C)$
	Open PS	$inters(\{w, z/A\}, B, C)$	$inters(\{w, z/A\}, \{1,2\}, C)$

lSet3	lSet2 / lSet1	Open PS
Unbound	Unbound	$inters(A, \{x, y/B\}, C)$
	Ground	$inters(\{1,2\}, \{x, y/B\}, C)$
	Open PS	$inters(\{w, z/A\}, \{x, y/B\}, C)$

Tabella 3.14: Rule 16-18 with lSet3 unbound

lSet3	lSet2 / lSet1	Unbound	Ground
Open PS	Unbound	$inters(A, B, \{u, v/C\})$	$inters(A, \{3,4\}, \{u, v/C\})$
	Ground	$inters(\{3,4\}, B, \{u, v/C\})$	$inters(\{3,4\}, \{3,4\}, \{u, v/C\})$
	Open PS	$inters(\{w, z/A\}, B, \{u, v/C\})$	$inters(\{w, z/A\}, \{3,4\}, \{u, v/C\})$

lSet3	lSet2 / lSet1	Open PS
Open PS	Unbound	$inters(A, \{x, y/B\}, \{u, v/C\})$
	Ground	$inters(\{3,4\}, \{x, y/B\}, \{u, v/C\})$
	Open PS	$inters(\{w, z/A\}, \{x, y/B\}, \{u, v/C\})$

Tabella 3.15: Rule 16-18 with lSet3 open non-ground

Riportiamo come esempio il programma di valutazione della regola 13.

```
@Test
public void testIntersRule1314() throws Failure {
```

```
    Solver solver = new Solver();

    // rule 13: {} inters lset = {}

    //unbound lSet2
    LSet S1 = LSet.empty().setName("S1");
    LSet S2 = new LSet().setName("S2");
    LSet S3 = new LSet().setName("S3");

    solver.solve(S1.inters(S2, S3));

    assertEquals("[]", solver.getConstraint().toString());
    assertEquals(S3, LSet.empty());
    assertFalse(solver.nextSolution());

    //ground lSet2
    S2 = LSet.empty().ins(1,2).setName("S2");
    S3 = new LSet().setName("S3");

    solver.solve(S1.inters(S2, S3));

    assertEquals("[]", solver.getConstraint().toString());
    assertEquals(S3, LSet.empty());
    assertFalse(solver.nextSolution());

    //partially specified (open) lSet2: es. {x,y|B} (x, y unbound)
    LVar x = new LVar("x");
    LVar y = new LVar("y");

    S2 = new LSet("B").ins(x,y).setName("S2");
    S3 = new LSet().setName("S3");

    solver.solve(S1.inters(S2, S3));

    assertEquals("[]", solver.getConstraint().toString());
    assertEquals(S3, LSet.empty());
    assertFalse(solver.nextSolution())


    // rule 14: lset inters {} = {}

    //...
}
```

Dove possibile è poi necessario verificare la correttezza anche nei corrispondenti casi false.

3.1.3 Test del vincolo diff

Presentiamo i casi di test della regola di differenza.

lSet3	lSet2 / lSet1	Unbound
Unbound	Unbound	$diff(A, A, C)$

Tabella 3.16: Rule extra

lSet3	lSet2 / lSet1	Unbound
Unbound	Unbound	$diff(A, B, C)$

Tabella 3.17: Rule 19

lSet3	lSet2 / lSet1	Unbound	Ground	Open PS
Unbound	$\emptyset$	$diff(\emptyset, B, C)$	$diff(\emptyset, \{1,2\}, C)$	$diff(\emptyset, \{x, y/B\}, C)$

Tabella 3.18: Rule 20

lSet3	lSet2 / lSet1	$\emptyset$
Unbound	Unbound	$diff(A, \emptyset, C)$
	Ground	$diff(\{1,2\}, \emptyset, C)$
	Open PS	$diff(\{x, y/A\}, \emptyset, C)$

Tabella 3.19: Rule 21

lSet3	lSet2 / lSet1	Unbound	Ground
$\emptyset$	Unbound	$diff(A, B, \emptyset)$	$diff(A, \{1,2\}, \emptyset)$
	Ground	$diff(\{1,2\}, B, \emptyset)$	$diff(\{1,2\}, \{1,2,3\}, \emptyset)$
	Open PS	$diff(\{w, z/A\}, B, \emptyset)$	$diff(\{w, z/A\}, \{1,2\}, \emptyset)$

lSet3	lSet2 / lSet1	Open PS
$\emptyset$	Unbound	$diff(A, \{x, y/B\}, \emptyset)$
	Ground	$diff(\{1,2\}, \{x, y/B\}, \emptyset)$
	Open PS	$diff(\{w, z/A\}, \{x, y/B\}, \emptyset)$

Tabella 3.20: Rule 22

lSet3	lSet2 / lSet1	Unbound	Ground
Unbound	Unbound	$diff(A, B, C)$	$diff(A, \{1,2\}, C)$
	Ground	$diff(\{1,2\}, B, C)$	$diff(\{1,2\}, \{2,3\}, C)$
	Open PS	$diff(\{w, z/A\}, B, C)$	$diff(\{w, z/A\}, \{1,2\}, C)$

lSet3	lSet2 / lSet1	Open PS
Unbound	Unbound	$diff(A, \{x, y/B\}, C)$
	Ground	$diff(\{1,2\}, \{x, y/B\}, C)$
	Open PS	$diff(\{w, z/A\}, \{x, y/B\}, C)$

Tabella 3.21: Rule 23-24-25 with lSet3 unbound

lSet3	lSet2 / lSet1	Unbound	Ground
Unbound	Unbound	$diff(A, B, \{1,2\})$	$diff(A, \{3,4\}, \{1,2\})$
	Ground	$diff(\{1,2,3,4\}, B, \{1,2\})$	$diff(\{1,2,3,4\}, \{3,4\}, \{1,2\})$
	Open PS	$diff(\{w, z/A\}, B, \{1,2\})$	$diff(\{w, z/A\}, \{3,4\}, \{1,2\})$

lSet3	lSet2 / lSet1	Open PS
Unbound	Unbound	$diff(A, \{x, y/B\}, \{1,2\})$
	Ground	$diff(\{1,2,3,4\}, \{x, y/B\}, \{1,2\})$
	Open PS	$diff(\{w, z/A\}, \{x, y/B\}, \{1,2\})$

Tabella 3.22: Rule 23-24-25 with lSet3 ground

lSet3	lSet2 / lSet1	Unbound	Ground
Unbound	Unbound	$diff(A, B, \{u, v/C\})$	$diff(A, \{3,4\}, \{u, v/C\})$
	Ground	$diff(\{1,2,3,4\}, B, \{u, v/C\})$	$diff(\{1,2,3,4\}, \{3,4\}, \{u, v/C\})$
	Open PS	$diff(\{w, z/A\}, B, \{u, v/C\})$	$diff(\{w, z/A\}, \{3,4\}, \{u, v/C\})$

lSet3	lSet2 / lSet1	Open PS
Unbound	Unbound	$diff(A, \{x, y/B\}, \{u, v/C\})$
	Ground	$diff(\{1,2,3,4\}, \{x, y/B\}, \{u, v/C\})$
	Open PS	$diff(\{w, z/A\}, \{x, y/B\}, \{u, v/C\})$

Tabella 3.23: Rule 23-24-25 with lSet3 non-ground

Riportiamo come esempio il codice di valutazione dei casi in cui il primo e il terzo insieme sono unbound.

```
private static int maxSolutionsTested = 10;

@Test
public void testDiffRule232425LSet3Unbound() throws Failure {
    Solver solver = new Solver();

    LSet S3 = new LSet("S3");
    IntLVar a = new IntLVar(1);
    IntLVar b = new IntLVar(2);

    //unbound lset1
    LSet S1 = new LSet("S1");

    //unbound lset2
    LSet S2 = new LSet("S2");

    solver.solve(S1.diff(S2, S3));

    assertEquals("diff(_S1,_S2,_S3)",
        solver.getConstraint().toString());
    assertFalse(solver.nextSolution());


    //ground lset2: es. {1,2}
    S1 = new LSet("S1");
    S2 = LSet.empty().ins(a,b).setName("S2");
    S3 = new LSet("S3");

    solver.clearStore();
    solver.solve(S1.diff(S2, S3));

    for (int i = 0; solver.nextSolution() && i <
        maxSolutionsTested; i++) {
      Solver s_aux = new Solver();
      s_aux.add(solver.getConstraint());

      assertTrue(s_aux.check(a.nin(S3).add(b.nin(S3))));
      assertEquals(S1.getTail(), S3.getTail());
    }


      //partially specified (open) lset2: es. {x,y|B} (x, y unbound)
```

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

    S1 = new LSet("S1");
    S2 = new LSet("B").ins(x,y).setName("S2");
    S3 = new LSet("S3");

    solver.clearStore();

    solver.solve(S1.diff(S2, S3));

    for (int i = 0; solver.nextSolution() && i <
        maxSolutionsTested; i++) {
      Solver s_aux = new Solver();
      s_aux.add(solver.getConstraint());

      assertTrue(s_aux.check(x.nin(S3).add(y.nin(S3))));

      Constraint diffTail = null;

      if(solver.getConstraint().getArg(1) instanceof Constraint)
        diffTail = (Constraint) solver.getConstraint().getArg(1);
      else
        diffTail = (Constraint) solver.getConstraint();
      assertEquals(diffTail.getArg(1), S1.getTail());
      assertEquals(diffTail.getArg(2), S2.getTail());
      assertEquals(diffTail.getArg(3), S3.getTail());
    }

    //...
}
```

Il metodo `getArg(i)` della classe `Constraint` se invocato su una congiunzione di vincoli:

- ritorna il primo vincolo se $i = 1$;
- ritorna i restanti vincoli della congiunzione se $i = 2$;
- ritorna `null` altrimenti.

Se invece viene invocato su un vincolo atomico, restituisce l'$i - esimo$ argomento se presente, `null` altrimenti.

Tramite questo metodo è quindi possibile verificare che il contenuto dello store del solver sia quello da noi atteso. Nell'esempio appena riportato, con questo metodo viene controllato che il primo vincolo nello store sia $diff(A, B, C)$, dove con A, B e C indichiamo le code di S_1, S_2 e S_3.

Dove possibile è poi necessario verificare la correttezza anche nei corrispondenti casi false.

3.1.4 Esempi per argomenti di tipo Set e CP

Come accennato nel capitolo 1, la libreria JSetL prevede la possibilità di avere oggetti di tipo `Set` e `CP` come argomenti secondari (`aConstraint.argument2` e/o `aConstraint.argument3`) dei vincoli insiemistici. Per questi casi ci limitiamo a riportare un programma di esempio.

```
public class TestSubsetCPandSet{
  //set case
  @Test
  public void testSubsetSet() throws Failure{
    Solver solver = new Solver();

    LSet S1 = LSet.empty().ins(1,2);

    Set<Integer> S2 = new TreeSet<>();
    S2.add(1);
    S2.add(2);
    S2.add(3);

    solver.solve(S1.subset(S2));

    assertEquals("[]", solver.getConstraint().toString());
  }

  //CP case
  @Test
  public void testSubsetCP() throws Failure{
    Solver solver = new Solver();

    LSet S1 = LSet.empty().ins(1,2);
    LSet S2 = LSet.empty().ins(3,4);
    LSet S3 = LSet.empty().ins(1,2,3);

    CP cp1 = new CP(S1, S2);
```

```
    CP cp2 = new CP(S3, S2);

    solver.solve(cp1.subset(cp2));

    assertEquals("[]", solver.getConstraint().toString());
  }

}
```

Analoghi sono i codici per inters e diff.

3.2 Valutazione di efficienza

Dopo aver ragionevolmente verificato che la nuova implementazione funzioni correttamente nei casi più comuni, procediamo con la valutazione della sua efficienza.

Il metodo adottato prevede di interrogare il solver con lo stesso programma, nella nuova e nella vecchia implementazione, per poi confrontare i tempi di esecuzione. Anche in questo caso, l'obiettivo è valutare un ventaglio di casi ragionevole, concentrandosi su quelli più comuni.

Più precisamente, come si può vedere nel codice sottostante, la tecnica prevede di risolvere lo stesso problema un certo numero n di volte, tipicamente n = 100, misurando, tramite la funzione di sistema `nanoTime()`, il tempo impiegato dal solver per restituire la prima soluzione.

Nell'esempio seguente viene misurato il tempo di risoluzione di 100 vincoli $S_1 \subseteq S_2$, con S_1 e S_2 insiemi ground. Da notare come prima di ogni risoluzione, le variabili unbound e non-ground vengano inizializzate di nuovo; questo viene fatto perchè la risoluzione dell'iterazione precedente potrebbe averne modificato il contenuto.

```
public class NonGroundSubsetNonGroundExtd {
  public static void main(String[] args) throws Failure {
    Solver solver = new Solver();

    long startTime = System.nanoTime();

    Integer[] elems =
        {25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};
    LSet A = new LSet("A");
    LSet B = new LSet("B");
```

```
    LSet S1 = A.insAll(elems).setName("S1");
    LSet S2 = B.insAll(elems).ins(100).setName("S2");

    S1.output();
    S2.output();

    for(int c=0;c<10;c++) {
      solver.clearStore();
      A = new LSet("A");
      B = new LSet("B");
      S1 = A.insAll(elems).setName("S1");
      S2 = B.insAll(elems).ins(100).setName("S2");

      solver.solve(S1.subset(S2));
    }

    long endTime  = System.nanoTime();
    long totalTime = endTime - startTime;
    System.out.println((float)totalTime/1000000000 + " sec");
   }
}
```

Effettuiamo 4 misurazioni per ogni programma di valutazione e calcoliamo i tempi medi. Nelle tabelle sottostanti con E indichiamo che il caso è banale, tipicamente perché tutti gli argomenti sono unbound o perché qualche argomento è l'insieme vuoto. Con ∞ si indicano tipicamente tempi superiori ai 3 minuti circa. Fa eccezione l'ultima tabella, in cui intendiamo tempi superiori ai 4 minuti circa.

I tipi di insiemi utilizzati sono riportati nella seguente tabella.

Dicitura	Insieme	Implementazione
Empty	$\emptyset$	`LSet.empty();`
Unbound	A	`new LSet("A");`
Ground	$\{a, ..., b\}, a < b, b - a \geq 10$	`LSet.empty().insAll(elems);`
Open Non Ground	$\{a, ..., b/A\}, a < b, b - a \geq 10$	`new LSet("A").insAll(elems);`

Fa eccezione il caso in cui il terzo argomento è open non-ground, in quel caso ci riferiamo ad un insieme del tipo $\{x, y/C\}$, la cui implementazione è `new LSet("C").ins(new LVar("x"), new LVar("y"));`.

3.2.1 Efficienza del vincolo di inclusione

Riportiamo nella tabella sottostante i risultati relativi al vincolo di inclusione.

SubsetExtd	Empty	Unbound	Ground	Open Non Ground
Exec: 100	T_m	T_m	T_m	T_m
Empty	E	E	0,122	E
Old	E	E	0,101	E
Unbound	0,055	E	1,236	48,837
Old	0,061	E	2,494	2,727
Ground	0,252	0,525	0,608	0,661
Old	0,259	0,190	0,191	5,195
Open NG	0,247	0,228	0,257	0,291
Old	0,233	0,921	∞	16,216

Tabella 3.24: Subset Times

Analizzando la tabella risulta evidente come, nel caso di primo insieme unbound e secondo insieme non-ground, la nuova implementazione performi notevolmente peggio di quella vecchia. La funzione responsabile della risoluzione di questo caso è `subsetRule3_7` e la corrispondente regola è:

$$\dot{A} \subseteq \{x \sqcup B\} \rightarrow A = \{x \sqcup N\} \wedge x \notin N \wedge N \subseteq B \bigvee x \notin A \wedge A \subseteq B \quad (2.7)$$

Osserviamo che il primo ramo impone che l'elemento x estratto da B sia contenuto in A, quindi applicando ripetutamente la regola otteniamo che la prima soluzione generata è quella in cui tutti gli elementi di B sono contenuti in A, ovvero $A = B$. Per lo stesso motivo la soluzione in cui A non contiene nessun'elemento di B sarà l'ultima generata.

Ora, se si considera il tempo complessivo in cui il solver genera tutte le soluzioni, l'ordine in cui queste vengono restituite è irrilevante. Spesso strumenti come JSetL vengono però utilizzati per verificare la soddisfacibilità di vincoli; per fare questo è sufficiente generare la prima soluzione. Concludiamo che è quindi vantaggioso che la prima soluzione venga generata nel più breve tempo possibile e che sia la più semplice.

Per questi motivi modifichiamo la regola astratta 2.7, invertendo i rami non-deterministici, nel seguente modo:

$$\dot{A} \subseteq \{x \sqcup B\} \rightarrow x \notin A \wedge A \subseteq B \bigvee A = \{x \sqcup N\} \wedge x \notin N \wedge N \subseteq B \quad (2.7)$$

Modifichiamo poi l'implementazione corrispondente e ripetiamo la valutazione di efficienza, ottenendo i seguenti risultati.

SubsetExtd	Empty	Unbound	Ground	Open Non Ground
Exec: 100	T_m	T_m	T_m	T_m
Empty	E	E	0,122	E
Old	E	E	0,101	E
Unbound	0,055	E	0,422	0,386
Old	0,061	E	2,494	2,727
Ground	0,252	0,525	0,608	0,661
Old	0,259	0,190	0,191	5,195
Open NG	0,247	0,228	0,257	0,291
Old	0,233	0,921	∞	16,216

Tabella 3.25: Subset Times

Il miglioramento è evidente. Ora gli unici due casi in cui la nuova implementazione è leggermente meno efficiente della vecchia sono quelli in cui il primo inisieme è ground e il secondo è ground o unbound.

Vista la bontà dei risultati ottenuti, prima di valutare le prestazioni degli altri vincoli, applichiamo lo stesso principio di inversione dei rami nondeterministici anche per le regole 2.18, 2.18 bis e 2.24. Le nuove regole sono riportate di seguito.

$$inters(A, \{x \sqcup B\}, \dot{C}) \rightarrow x \notin A \wedge inters(A, B, C) \bigvee A = \{x \sqcup N\} \wedge x \notin N \wedge C = \{x \sqcup N_1\} \wedge x \notin N_1 \wedge inters(N, B, N_1) \quad (2.18)$$

$$inters(\{x \sqcup A\}, \dot{B}, \dot{C}) \rightarrow x \notin B \wedge inters(A, B, C) \bigvee B = \{x \sqcup N\} \wedge x \notin N \wedge C = \{x \sqcup N_1\} \wedge x \notin N_1 \wedge inters(A, N, N_1) \quad (2.18 \text{ bis})$$

$$diff(A, \{x \sqcup B\}, C) \rightarrow x \notin A \wedge diff(A, B, C) \bigvee A = \{x \cup N\} \wedge x \notin N \wedge diff(N, B, C) \quad (2.24)$$

3.2.2 Efficienza del vincolo di intersezione

Riportiamo nella tabella sottostante i risultati relativi al vincolo di intersezione.

IntersExtd		Unbound	Ground	Open Non Ground
Exec: 100	Result Set	T_m	T_m	T_m
Unbound	Unbound	E	0,098	0,119
	Old	E	17,930	4,658
	Ground	0,845	1,218	1,091
	Old	0,199	2,645	2,637
	Open NG	0,023	0,174	0,136
	Old	0,250	12,001	5,149
Ground	Unbound	0,117	0,354	2,773
	Old	18,367	∞	∞
	Ground	1,152	1,286	1,548
	Old	2,579	1,220	4,385
	Open NG	0,154	∞	2,576
	Old	12,645	∞	∞
Open Non Ground	Unbound	0,113	0,334	2,473
	Old	4,370	∞	∞
	Ground	1,519	1,069	1,954
	Old	2,584	1,313	6,741
	Open NG	0,151	∞	∞
	Old	4,881	∞	16,216

Tabella 3.26: Inters Times

Osservando i risultati notiamo che la nuova implementazione performa alla pari o meglio in tutti i casi tranne quello in cui i primi due insiemi sono unbound e il terzo è ground. Sono presenti però due ∞ nei casi in cui il primo argomento è ground o non-ground, il secondo è ground e il risultato non-ground; ovvero $inters(\{a, ..., b\}, \{c, ..., d\}, \{x, y/C\})$ e $inters(\{a, ..., b/A\}, \{c, ..., d\}, \{x, y/C\})$.

Questo casi sono tutti gestiti dalla regola 16:

$$\begin{aligned} inters(A, B, \{x \sqcup C\}) \rightarrow \\ A = \{x \sqcup N_1\} \wedge B = \{x \sqcup N_2\} \wedge inters(N_1, N_2, C) \end{aligned} \tag{2.16}$$

Avendo però $\{x, y/C\}$ solo due elementi, dopo due applicazioni della suddetta regola, si passa ad applicare la regola 18:

$$inters(A, \{x \sqcup B\}, \dot{C}) \rightarrow x \notin A \wedge inters(A, B, C) \bigvee A = \{x \sqcup N\} \wedge x \notin N \wedge C = \{x \sqcup N_1\} \wedge x \notin N_1 \wedge inters(N, B, N_1) \quad (2.18)$$

La regola su cui incentrare futuri sforzi di ottimizzazione dovrebbe quindi essere la 18.

3.2.3 Efficienza del vincolo di differenza

Riportiamo nella tabella sottostante i risultati relativi al vincolo di differenza.

DiffExtd		Unbound	Ground	Open Non Ground
Exec: 100	Result Set	T_m	T_m	T_m
Unbound	Unbound	E	0,031	0,016
	Old	E	0,700	0,152
	Ground	0,423	0,808	0,404
	Old	2,977	∞	3,976
	Open NG	0,024	0,111	0,044
	Old	0,103	1,264	0,381
Ground	Unbound	0,038	192,721	119,626
	Old	0,541	∞	1,530
	Ground	2,185	1,070	0,925
	Old	∞	0,540	∞
	Open NG	0,138	175,770	105,996
	Old	0,788	∞	∞
Open Non Ground	Unbound	0,082	3,606	0,267
	Old	1,293	∞	2,294
	Ground	0,559	∞	0,714
	Old	∞	∞	∞
	Open NG	0,138	2,555	0,326
	Old	1,705	∞	2,729

Tabella 3.27: Diff Times

Osservando i risultati notiamo che la nuova implementazione performa alla pari o meglio in tutti i casi tranne quello in cui il primo insieme è ground, il

secondo open non-ground e il risultato unbound. Più in generale i casi in cui i tempi di esecuzione sono alti sono i seguenti:

- $diff(\{a, ..., b\}, \{c, ..., d\}, C)$
- $diff(\{a, ..., b\}, \{c, ..., d/B\}, C)$
- $diff(\{a, ..., b\}, \{c, ..., d\}, \{x, y/C\})$
- $diff(\{a, ..., b\}, \{c, ..., d/B\}, \{x, y/C\})$

Conduciamo ora un ragionamento analogo a quanto fatto in precedenza: i primi due casi rientrano nella regola 24:

$$\begin{aligned} diff(A, \{x \sqcup B\}, C) \rightarrow x \notin A \wedge diff(A, B, C) \\ \bigvee \\ A = \{x \cup N\} \wedge x \notin N \wedge diff(N, B, C) \end{aligned} \tag{2.24}$$

mentre i secondi due nella regola 23:

$$diff(A, B, \{x \sqcup C\}) \rightarrow A = \{x \sqcup N\} \wedge x \notin N \wedge x \notin B \wedge diff(N, B, C) \tag{2.23}$$

Avendo però $\{x, y/C\}$ solo due elementi, dopo due applicazioni della suddetta regola, si passa ad applicare la regola 24, come nel caso precedente. La regola su cui incentrare futuri sforzi di ottimizzazione dovrebbe quindi essere la 24.

Capitolo 4

Casi speciali

I casi in cui due o più argomenti di un vincolo hanno la stessa parte resto, che indicheremo come *casi speciali*, possono causare un errato funzionamento delle regole di riscrittura standard. Tipicamente causano la *non-terminazione* della procedura di risoluzione, come si può verificare eseguendo il seguente codice d'esempio:

```
//inters({s1...sm|X},{t1...tn/X},{u1...uk|X})
public class IntersTailTest2 {
  public static void main(String[] args) throws Failure{
    Solver solver = new Solver();

    LSet X = new LSet("X");
    LSet S1 = X.ins(2).setName("S1");
    LSet S2 = X.ins(2).setName("S2");
    LSet S3 = X.ins(2).setName("S3");

    solver.solve(S1.inters(S2,S3));
    do {
      S1.output();
      S2.output();
      solver.showStore();
    } while (solver.nextSolution());

  }
}
```

Per ottenere la terminazione è quindi necessario aggiungere altre regole ad-hoc.

4.1 Regole di riscrittura astratte

In prima approssimazione tutti i casi speciali potrebbero essere risolti semplicemente tramite regole che trattano il vincolo come derivato, riscrivendolo in una congiunzione di altri vincoli primitivi. In tutte le regole riportate la coda degli insiemi è indicata con X.

$$subset(X, \{t_1...t_n/X\}), Xvar \rightarrow un(X, \{t_1...t_n/X\}, \{t_1...t_n/X\}) \tag{4.1}$$

$$\begin{aligned} subset(\{t_1...t_n/X\}, \{s_1...s_m/X\}) \rightarrow \\ un(\{t_1...t_n/X\}, \{s_1...s_m/X\}, \{s_1...s_m/X\}) \end{aligned} \tag{4.2}$$

$$\begin{aligned} inters(X, \{t_1...t_n/X\}, S3), S3var \rightarrow \\ un(D, S3, X), un(E, S3, \{t_1...t_n/X\}), disj(D, E) \end{aligned} \tag{4.3}$$

$$\begin{aligned} inters(\{s_1...s_m/X\}, \{t_1...t_n/X\}, \{u_1...u_k/X\}) \rightarrow \\ un(D, \{u_1...u_k/X\}, \{s_1...s_m/X\}), un(E, \{u_1...u_k/X\}, \{t_1...t_n/X\}), disj(D, E) \end{aligned} \tag{4.4}$$

$$\begin{aligned} inters(X, \{t_1...t_n/X\}, \{u_1...u_k/X\}) \rightarrow \\ un(D, \{u_1...u_k/X\}, X), un(E, \{u_1...u_k/X\}, \{t_1...t_n/X\}), disj(D, E) \end{aligned} \tag{4.5}$$

$$\begin{aligned} inters(\{s_1...s_m/X\}, X, \{u_1...u_k/X\}) \rightarrow \\ un(D, \{u_1...u_k/X\}, \{s_1...s_m/X\}), un(E, \{u_1...u_k/X\}, X), disj(D, E) \end{aligned} \tag{4.6}$$

$$\begin{aligned} inters(\{s_1...s_m/X\}, \{t_1...t_n/X\}, X) \rightarrow \\ un(D, X, \{s_1...s_m/X\}), un(E, X, \{t_1...t_n/X\}), disj(D, E) \end{aligned} \tag{4.7}$$

$$\begin{aligned} diff(X, \{t_1...t_n/X\}, C) \rightarrow \\ subset(C, X), un(\{t_1...t_n/X\}, C, D), subset(X, D), disj(\{t_1...t_n/X\}, C) \end{aligned} \tag{4.8}$$

$$\begin{aligned} diff(\{s_1...s_m/X\}, \{t_1...t_n/X\}, C) \rightarrow \\ subset(C, \{s_1...s_m/X\}), un(\{t_1...t_n/X\}, C, D), \\ subset(\{s_1...s_m/X\}, D), disj(\{t_1...t_n/X\}, C) \end{aligned} \tag{4.9}$$

4.2 Implementazione dei casi speciali

Come detto all'inizio di questo capitolo, l'esecuzione di vincoli che rientrano nei casi speciali può causare un comportamento errato, ma potrebbe anche concludersi in modo corretto; prima di implementare le regole per i casi speciali è quindi necessario verificarne l'effettiva necessità. Effettuiamo quindi dei test che prevedono di verificare il comportamento del solver in presenza dei suddetti casi.

Si riportano qui i codici di test per la regola `subset`, analoghi sono i codici per `inters` e `diff`.

```
//subset(X,{t1...tn/X}), X
public class SubsetTailTest1 {

  public static void main(String[] args) throws Failure{
    Solver solver = new Solver();

    LSet X = new LSet("X");
    LSet S2 = X.ins(1,2).setName("S2");

    solver.solve(X.subset(S2));

    do {
      X.output();
      S2.output();
      solver.showStore();
      System.out.println("\n");
    } while (solver.nextSolution());

  }

}
```

```
//subset({t1...tn|X},{s1...sm|X})
public class SubsetTailTest2{

  public static void main(String[] args) throws Failure{
    Solver solver = new Solver();

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

```
    LSet S1 = X.ins(5,2,3,4).setName("S1");
    LSet S2 = X.ins(1,2,3,4,5,6).setName("S2");

    solver.solve(S1.subset(S2));

    do {
      S1.output();
      S2.output();
      solver.showStore();
      System.out.println("\n");
    } while (solver.nextSolution());

  }

}
```

Eseguendo il primo test, si ottengono numerose soluzioni, ne riportiamo alcune:

```
_X = {1,2/_N6}
_S2 = {1,2/_N6}
Store: 1 nin _N6 AND 2 nin _N6 AND 1 nin _N6 AND 2 nin _N6


_X = {1/_N2}
_S2 = {1,2/_N2}
Store: 1 nin _N2 AND 2 nin _N2 AND 1 nin _N2


_X = {2/_N29}
_S2 = {1,2/_N29}
Store: 1 nin _N29 AND 2 nin _N29 AND 2 nin _N29


_X = unknown
_S2 = {1,2/_X}
Store: 1 nin _X AND 2 nin _X
```

Il solver si è comportato in modo corretto, ma ha generato tutte le soluzioni. Analizzando il caso astratto $X \subseteq \{t_1...t_n/X\}$, si nota come la risoluzione potrebbe terminare semplicemente restituendo true. Questo permette di in-

trodurre il secondo motivo per implementare le regole per i casi speciali, ovvero l'efficienza e la semplicità di lettura delle soluzioni.
Procediamo quindi modificando la regola 4.1 nel seguente modo:

$$subset(X, \{t_1...t_n/X\}), Xvar \rightarrow true \tag{4.10}$$

La sua implementazione in JSetL sarà:

```
// subset(X,{t1...tn/X}), X var --> true
private boolean subsetTailRule1(@NotNull LSet lSet1, @NotNull LSet
    lSet2, @NotNull AConstraint aConstraint) {
    /*
    * assert varie
    */

    if (lSet1 == lSet2.getTail()) {
      aConstraint.setSolved(true);

      return true;
    }

    return false;
}
```

Rieseguendo il test, l'esecuzione termina subito come ci aspettiamo:

```
_X = unknown
_S2 = {1,2/_X}
Store: []
```

Per la seconda regola della subset (4.11), applichiamo un ragionamento diverso: osservando la testa della regola $subset(\{t_1...t_n/X\}, \{s_1...s_m/X\})$, notiamo che la sua esecuzione con le regole standard, ricade nel caso 2.6, che riportiamo di seguito:

$$\begin{array}{c} \{x|A\} \subseteq \{y|B\} \rightarrow x = y \wedge A \subseteq \{y|B\} \\ \bigvee \\ x \neq y \wedge x \in B \wedge A \subseteq \{y|B\} \end{array} \tag{2.6}$$

Intuitivamente, quello che fa questa regola, è estrarre il primo elemento del primo insieme, imporre che sia contenuto nel secondo insieme e ripetere il

procedimento. Applicando questo ragionamento al caso di nostro interesse è facile verificare che si arriva ad un vincolo del tipo $subset(X, \{s_1...s_m/X\})$ che è esattamente la testa della regola per il primo caso speciale dalla subset.

Concludiamo quindi che è sufficiente implementare quest'ultima, per ottenere il comportamento desiderato in entrambi i casi.
Eseguiamo il secondo test e otteniamo il risultato atteso:

```
_S1 = {5,2,3,4/_X}
_S2 = {1,2,3,4,5,6/_X}
Store: []
```

Riguardo ai casi speciali per il vincolo inters, se osserviamo le regole 4.4, 4.5, 4.6, notiamo che, a livello implementativo, si traducono nello stesso modo, ovvero:

$$inters(S_1, S_2, S_3) \rightarrow un(D, S_3, S_1), un(E, S_3, S_2), disj(D, E) \tag{4.11}$$

Decidiamo quindi di implementare un'unica regola per tutti e tre i casi:

```
private boolean intersTailRule234(@NotNull LSet lSet1, @NotNull
    LSet lSet2, @NotNull LSet lSet3, @NotNull AConstraint
    aConstraint) {
    /*
     * assert varie
     */

    if (lSet3.isBound() && //exclude intersTailRule5 case:
        inters({s1...sm|X},{t1...tn/X},X)
      (lSet1.getTail() == lSet2.getTail() && lSet2.getTail() ==
          lSet3.getTail())) {

      LSet D = new LSet("D");
      LSet E = new LSet("E");

      AConstraint unioConstraint1 = new
          AConstraint(Environment.unionCode, D, lSet3, lSet1);
      AConstraint unionConstraint2 = new
          AConstraint(Environment.unionCode, E, lSet3, lSet2);

      solver.add(unioConstraint1);
      solver.add(unionConstraint2);
```

```
      aConstraint.argument1 = D;
      aConstraint.argument2 = E;
      aConstraint.argument3 = null;
      aConstraint.constraintKindCode = Environment.disjCode;

      solver.storeUnchanged = false;

      return true;
      }

    return false;
}
```

Si noti che per come è definita la funzione `getTail()` in JSetL, anche le condizioni di ingresso possono essere racchiuse in una sola; infatti dato il seguente codice:

```
LSet X = new LSet("X");
LSet Y = X.ins(1,2,3).setName("Y");

X.getTail();
Y.getTail();
```

in entrambi i casi l'esecuzione di getTail() restituisce X.

In conclusione, eseguendo i test per ogni caso speciale, troviamo che le regole per cui è necessaria l'implementazione sono:

- subsetTailRule1 (4.1);
- intersTailRule2 (4.4), intersTailRule3(4.5), intersTailRule4(4.6);
- diffTailRule1(4.8), diffTailRule2(4.9);

mentre le regole per cui non è necessaria l'implementazione sono:

- subsetTailRule2(4.2);
- intersTailRule1(4.3), intersTailRule5(4.9).

Infine inseriamo le chiamate ai metodi nel metodo corrispondente `nomeVincoloRule`, facendo in modo che vengano eseguite prima delle chiamate ai metodi per le regole standard che trattano gli stessi casi. Ad esempio la

regola `subsetTailRule1` rientra nel caso più generale in cui il primo insieme è unbound e il secondo bound e deve quindi precedere la chiamata alla regola `subsetRule3_7`.

La nuova `subsetRule` è quindi la seguente:

```
private void subsetRule(@NotNull LSet lSet1, @NotNull LSet lSet2,
    @NotNull AConstraint aConstraint) {
    /*
     * assert varie
     */

        if (subsetRuleExtra(lSet1, lSet2, aConstraint)) return;
        else if (subsetRule1(lSet1, lSet2, aConstraint)) return;
        else if (subsetRule2(lSet1, lSet2, aConstraint)) return;
        else if (subsetTailRule1(lSet1, lSet2, aConstraint)) return;
        else if (subsetRule3_7(lSet1, lSet2, aConstraint)) return;
        else if (subsetRule4(lSet1, lSet2, aConstraint)) return;
        else if (subsetRule5(lSet1, lSet2, aConstraint)) return;
        else if (subsetRule6(lSet1, lSet2, aConstraint)) return;
        else return;

}
```

Analogamente deve essere fatto per `intersRule` e `diffRule`.

4.3 NEQ-Elimination

In JSetL, per effettuare la risoluzione, il solver riscrive ripetutamente i vincoli presenti nel proprio store, fino a raggiungere una forma irriducibile, detta *solved form*. La solved form è una congiunzione di vincoli che ha la proprietà di essere soddisfacibile, ovvero esiste una sostituzione che la rende vera, e spesso questa sostituzione è il l'insieme vuoto.
Ad esempio:

$$un(X, Y, Z) \wedge disj(X, Z) \tag{4.12}$$

è una solved form e risulta soddisfacibile per $X = \emptyset$.

In realtà la solved form è sicuramente soddisfacibile soltanto se nello store non sono presenti vincoli di disuguaglianza sulle variabili, ad esempio $X \neq \emptyset$. Risulta evidente come aggiungendo questo secondo vincolo, l'esempio precedente risulti insoddisfacibile.

Per accorgersi di questo, una volta terminata la riscrittura, il solver lancia una procedura, detta *neq-elimination*, con l'obiettivo di individuare vincoli di disuguaglianza (neq) su variabili insiemistiche; se la ricerca ha successo, assegna a queste variabili un valore opportuno e ricontrolla l'intero vincolo.

4.3.1 Implementazione della NEQ-Elimination

In JSetL, ogni volta che si trasforma un vincolo derivato in vincolo primitivo o si introduce un nuovo vincolo primitivo, che prevede una forma risolta contenente una o più variabili insiemistiche, è necessario modificare la procedura di neq-elimination in modo che gestisca anche i nuovi vincoli.

Per ottenere questo, da un punto di vista implementativo è sufficiente modificare, all'interno della classe `NeqRemover`, la variabile privata statica `unsafeSolvedFormsConstraintKinds`, aggiungendo i codici `Environment` dei vincoli desiderati.

```
private static final Set<Integer> unsafeSolvedFormsConstraintKinds
    = new HashSet<>(Arrays.asList(
  Environment.eqCode,
  Environment.unionCode,
  Environment.subsetUnionCode,
  Environment.unionSubsetCode,
  Environment.brCompCode,
  Environment.subsetCompCode,
  Environment.compSubsetCode,
  Environment.pfCompCode,
  Environment.brRanCode,
  Environment.pfRanCode,
  Environment.brDomCode,
  Environment.pfDomCode,
  Environment.invCode,
  Environment.idCode,
  Environment.subsetCode,
  Environment.intersCode,
  Environment.diffCode
  //Aggiungere qui i codici dei vincoli desiderati
  )
);
```

4.3.2 Valutazione della NEQ-Elimination

Dopo questa semplice modifica verifichiamo che il solver applichi effettivamente la procedura di neq-elimination ai nuovi vincoli. Per fare questo sottoponiamo una congiunzione di vincoli irriducibile, contenente almeno un vincolo di disuguaglianza.

```
public class TestSubsetNeqElimination {

  @Test
  public void testSubsetNeq() throws Failure{
    Solver solver = new Solver();

    LSet S1 = new LSet("S1");
    LSet S2 = new LSet("S2");

    solver.solve(S1.subset(S2).add(S1.disj(S2).add(S1.neq(LSet.empty()))));

    solver.showStore();
  }

}
```

Al termine della risoluzione:

- se lo store contiene la stessa congiunzione di vincoli di partenza, la procedura non è stata attivata;

  ```
  Store: subset(_S1,_S2) AND disj(_S1,_S2) AND _S1 neq {}
  ```

- mentre lo è stata in tutti gli altri casi; nel nostro esempio l'esecuzione termina lanciando l'eccezione `Failure`, in quanto sostituendo l'insieme vuoto a S_1, la congiunzione di vincoli diventa insoddisfacibile.

I codici di prova per inters e diff sono analoghi.

```
public class TestInterstNeqElimination{

  @Test
  public void test() throws Failure{
```

```
    Solver solver = new Solver();

    LSet S1 = new LSet("S1");
    LSet S2 = new LSet("S2");
    LSet S3 = new LSet("S3");

    solver.add(S1.inters(S2,S3));
    solver.add(S1.disj(S3));
    solver.add(S3.neq(LSet.empty()));

    assertFalse(solver.check());

  }

}
```

```
public class TestDiffNeqElimination {

  @Test
  public void test() {
    Solver solver = new Solver();

    LSet S1 = new LSet("S1");
    LSet S2 = new LSet("S2");
    LSet S3 = new LSet("S3");

    solver.add(S1.diff(S2,S3));
    solver.add(S1.disj(S3));
    solver.add(S3.neq(LSet.empty()));

    assertFalse(solver.check());
  }

}
```

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi è stato mostrato come trasformare alcuni vincoli insiemistici derivati in vincoli primitivi, all'interno della libreria Java JSetL. Per l'implementazione e la valutazione dei nuovi vincoli si è cercato di definire una casistica il più possibile ampia, individuando regole specifiche che favorissero l'efficienza.

I risultati della valutazione dei tempi, presentati nel paragrafo 3.2, mostrano come questo abbia permesso di migliorare significativamente le prestazioni nel caso di vincoli insiemistici applicati a insiemi non specificati (unbound) o parzialmente specificati (partially specified bounded/unbounded). Nei casi di insiemi completamente specificati (ground) le prestazioni sono invece spesso simili a quelle dell'implementazione con vincoli derivati.

Restano però casi isolati in cui le prestazioni non sono soddisfacenti, come nei casi evidenziati nei paragrafi 3.2.2 e 3.2.3. Nell'ottica di migliorare ancora le prestazioni, sarebbe quindi opportuno realizzare regole migliori, che vadano a sostituire, completamente o in parte, le attuali regole 18 e 24.

In conclusione, la traduzione dei vincoli derivati in vincoli primitivi può portare notevoli miglioramenti in termini di efficienza temporale; lo sviluppo più naturale consiste quindi nell'applicare questa strategia anche agli altri vincoli derivati presenti in JSetL.

Un ulteriore sviluppo potrebbe essere un'analisi più approfondita delle regole di riscrittura dei casi speciali (capitolo 4) che in questa tesi sono stati semplicemente trattati come vincoli derivati, salvo qualche eccezione.

Nello stesso ambito sarebbe poi utile un'analisi più approfondita anche delle motivazioni per cui applicare regole speciali, non limitandosi alla presenza di esecuzioni infinite, ma ad esempio prendendo in considerazione l'efficienza d'esecuzione. Sarebbe inoltre utile valutare queste regole alla luce delle regole "standard", con l'obiettivo di individuare riscritture superflue, come fatto ad esempio con le regole speciali per la subset(sottocapitolo 4.2).

Il terzo e ultimo sviluppo riguarda il testing. In questa tesi è stata infatti effettuata solo una valutazione approssimativa della correttezza delle regole implementate; può essere quindi utile un'attività di testing che sia ampia, ovvero ricopra tutto il panorama dei casi possibili, e che sia profonda, ovvero provveda a verificare la correttezza di tutte le soluzioni generate dal solver. Analogamente, per quanto riguarda l'efficienza, sarebbe utile misurare il tempo in cui il solver genera tutte le soluzioni possibili, non limitandosi alla prima.

Per quanto riguarda l'inserimento definitivo in JSetL dell'implementazione presentata ci sono principalmente due strade: la prima prevede di modificare `RwRulesSet`, inserendo le nuove implementazioni al posto di quelle preesistenti; la seconda prevede di mantenere l'implementazione nella classe `RwRulesSetExtd`, con lo svantaggio che ogni nuova estensione di `RwRulesSet` dovrà essere inserita in `RwRulesSetExtd`. Questo perché, come si evince dalla modifica alla classe `Solver` effettuata nel capitolo 2, in caso di più classi derivate, solo una potrebbe essere passata al solver stesso. Si noti infine che scegliendo la prima soluzione, sarebbe invece necessario ripristinare la modifica alla classe `Solver`.

Riferimenti bibliografici

[1] Agostino Dovier, Carla Piazza, Enrico Pontelli and Gianfranco Rossi
Sets and Constraint Logic Programming, ACM Transaction on Programming Language and Systems, Vol.22(5),Sept.2000, 861-931.

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

[3] Gianfranco Rossi, Roberto Amadini and Andrea Fois
JSetL User's Manual
http://www.clpset.unipr.it/jsetl/downloads/jsetl-3-0-manual.pdf

[4] JSetL Home Page
http://www.clpset.unipr.it/jsetl/

[5] Maximiliano Cristiá, Gianfranco Rossi
Rewrite Rules for $\subseteq$ and $\bigcap$
March 26, 2020